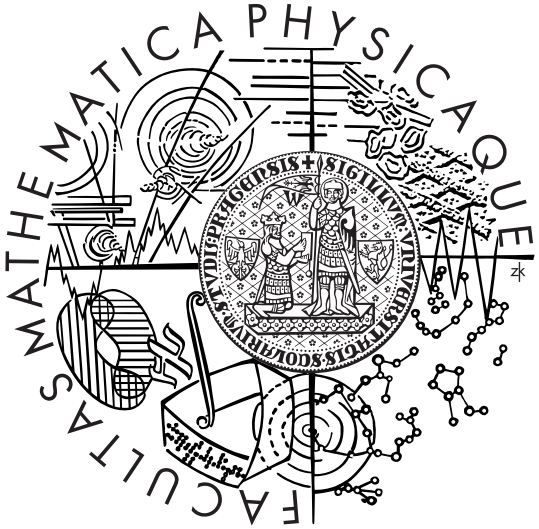
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Rodrigo Baquero

Deployment of SOFA 2 applications for the LeJOS platform

Department of Software Engineering

Supervisor: RNDr. Petr Hnetynka, Ph.D.

Study programme: Informatics

Specialization: Software systems

Prague 2012

Dedication:

I dedicate this thesis to my family, specially my parents Jairo and Martha. This work was possible thanks to their support.

Acknowledgments:

I would like to thank RNDr. Petr Hnetynka, Ph.D. for the advise, help and time he dedicated to this work.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on December 7, 2012 Rodrigo Baquero

Název práce: Nasazení SOFA 2 aplikací na platformě LeJOS
Autor: Rodrigo Baquero
Katedra: Katedra softwaroveho inzenyrstvi
Vedoucí diplomové práce: RNDr. Petr Hnetynka, Ph.D.
Abstrakt:

Vestavěné systémy jsou v naší společnosti všudypřítomné. Ovládájí vozidla, letadla a lékařské nástroje. Některé z těchto systémů jsou distribuované. Jsou součástí sítě a jejich činnost je koordionvána. Vývoj softwaru pro takovéto systémy může s sebou nést problémy.

V této práci navrhneme komponentový systém založený na SOFA 2, určený k zjednodušení vývoje software pro distribuované, vestavěné systémy, kde rozvržení jednotlivých součástí sítě je spravováno výhradně tímto komponentovým systémem. Jako model pro distribuované, vestavěné systémy použijeme Lego Mindstorms.

Navrhovaný přístup prezentuje bezvadnou distribuci komponent, nicméně pro plné využití potenciálu komponentového systému musí být v implemntaci zadáný nefunkcionální požadavky jako paměť, velikost programu nebo velikost disku.

Klíčová slova: Komponentové systémy, distribuované vestavěné zařízení, softwarové konektory

Title: Deployment of SOFA 2 applications for the LeJOS platform
Author: Rodrigo Baquero
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnetynka, Ph.D.
Abstract:

Embedded systems are ubiquitous in our society, they control vehicles, aircrafts and medical instruments. Some of these systems are distributed, which means they are part of a network and their operation is coordinated. Software development for such systems can be a difficult problem.

In this thesis we propose SOFA 2 component system to simplify the software development for distributed embedded systems where the distribution of components is handled entirely by the component system. Lego Mindstorms is proposed as the model of a distributed embedded system. A runtime environment for SOFA 2 and a demo application were developed to evaluate the approach.

The proposed approach delivers seamless component distribution. Nevertheless, non-functional requirements such as memory, program size or disk space must be included in the implementation to fully benefit from a component system.

Keywords: component systems, distributed embedded devices, software connectors

Contents

List of Figures

1. Introduction

The digital revolution has created affordable, mass produced and miniaturized electronic devices. For instance, in 1977 automobiles were mainly controlled by mechanical devices with no electronic microprocessors on board, today even a low end model can have up to 30 microprocessors which execute at least 10M lines of computer instructions. It is estimated that for a premium car, software development represents up to 15% of the value of the vehicle and half of warranty costs faced by car manufacturers are due to faulty electronics or software.

The automobile industry is not the only one to rely on electronics, embedded systems are now ubiquitous in aeronautics, medical equipment and consumer electronics [57]. By embedded systems we understand a system which is enclosed in another system i.e. the antilock braking system of a vehicle.

The challenge faced by software designers that target distributed embedded systems is to implement reliable and effective software solutions to be able to coordinate all the electronic systems, sensors and microprocessors. By distributed we mean a system whose elements use a computer network to coordinate their actions. This work focuses on the techniques to design and implement software for distributed embedded systems.

One of the alternatives to build software for such systems is to use a component system. A component can be defined as a "black-box entity with well defined interfaces and behavior" [1]. Components encapsulate common functionality hence, they help to emphasize the separation of concerns. A well defined interface means that a component can be exchanged from another or reused in a different application. Software is built by joining components. A component system is the realization of a component model. The model specifies the features and behavior of components and defines the rules to assemble them.

Software distribution is achieved by deploying software components to multiple devices. The component system supports such distribution by providing communication mechanisms to exchange data between components deployed either in the same device or deployed to different devices.

Since this work is not focused on hardware design, the Lego Mindstorms robotics kit is used as a hardware model for a distributed embedded system. This robotics kit is widely used in education and let designers create fast prototypes in the industry. It comes with a programmable micro controller called NXT.

Lego Mindstorms provides a component oriented solution to program NXT devices, it uses a language called NXT-G [61]. Lego bundles graphical programming environment with all the tools needed to create applications. In NXT-G the application is composed of blocks that resemble resemble its physical counterparts. Users can create applications by dragging blocks into a working space and connecting those blocks together. Lego developed this tool in conjunction with

National Instruments. The NXT-G language and programming environment is targeted at kids nevertheless is used by hobbyist and academics. National Instruments also created a special version of LabVIEW, a development environment widely used in control engineering for Lego Mindstorms.

Current component systems used to develop software for the Lego Mindstorms offer limited component distribution. They provide mechanisms to exchange data between devices but the component system do not takes care the communication between components. The responsibility of the exchange of data between components is transferred to the software designer.

In this thesis an existing component system will be adapted work with Lego Mindstorms to provide seamless component distribution. By seamless we mean the component system is responsible for the component distribution and the exchange of data between components is automated. Thus, software developers write less code and they can focus on solving their business needs.

In particular, the SOFA2 component system is used. SOFA2 supports the deployment of distributed applications to desktop computers that are able to run the Java Standard Edition (Java SE). SOFA2 also offers limited support for mobile devices that run Java Micro Edition (Java ME). In this platform is possible to deploy applications without distribution. Both Java SE and Java ME are based on the Java programming language.

This thesis uses a programming language with Java syntax to create applications for Lego NXT devices. The open source project LeJOS was identified as one of the best alternatives, it provides all the tools needed to create and execute programs in NXT devices. Also, by choosing a programming language in the NXT device which shares the same syntax as the implementation of SOFA 2 the code that can be reused in the NXT platform is maximized and software developers benefit from object oriented programming.

A new runtime environment of SOFA2 for Lego Mindstorms based on the existing work on mobile devices is proposed. This runtime environment supports the execution of SOFA2 components on NXT devices. It also contains a new communication middleware based on LeJOS to support the exchange of data between applications up to 4 NXT devices.

Goal of the thesis

The goal of this thesis consist in extending the SOFA 2 component system to support seamless deployment of distributed applications to multiple Lego Mindstorms NXT devices which are flashed with the custom LeJOS firmware.

The work includes an analysis of the characteristics of SOFA 2 and the specific constraints and features of the embedded device and LeJOS Virtual Machine to determine the characteristics of a new runtime environment for SOFA 2. The analysis must also identify changes in the development process. The implemen-

tation must include software development tools, a runtime environment of SOFA 2 for LeJOS and a demo application.

Structure of the text

This work is organized as follows: *Section 1* introduces Lego Mindstorms NXT device, its features, hardware characteristics, runtime environment, development tools and the LeJOS virtual machine. Also SOFA 2 component model is introduced. *Section 2* presents the proposed design and the analysis to support SOFA 2 runtime in LeJOS and a middleware to support distribution. *Section 3* describes the implementation of SOFA 2 for LeJOS and analyses its limitations. *Section 4* describes the development process in SOFA 2 for LeJOS and shows a demo application and it's architecture. It also evaluates the limitations of SOFA 2 for LeJOS, proposes solutions for these limitations and proposes areas for future development or research. In *Section 5* other component models that support distributed applications for embedded devices are analyzed and compared to SOFA 2 for LeJOS. The *Conclusion* section presents a summary of this work and revises the fulfillment of its goals.

2. Background

This chapter makes a brief introduction of the background technology, models and software required for this thesis. It starts by a description of the Lego Mindstorms platform and its communication features. It follows by by a introduction to Java Micro Edition JME and some techniques of code optimization frequently used in projects that target embedded devices. It ends with an introduction to SOFA 2 component system and its current implementation.

2.1 Lego Mindstorms NXT

Lego Mindstorms NXT is a robotics kit manufactured by Lego in conjunction with MIT Media Lab [28]. It comes with a programming environment based on a graphical component model and users can program of robots with no technical background. It was originally designed for children but soon it gathered interest among hobbyists, educators and professionals. [60]

2.1.1 NXT Hardware

The hardware included with this robotic kit is described below:

NXT Intelligent Brick [29] Is the brain of the robotics kit. It contains a microprocessor, bluetooth support, a LCD screen and ports to attach sensors or servo motors. Users can create programs and upload them to the NXT brick.

Touch sensor Contains a switch that detects whether it was pressed, released or bumped (pressed followed by a release).

Sound sensor Detects sounds in the environment up to 90 dB. Data is displayed as a percentage. Total silence is given the value of 0.

Light sensor Distinguishes between light and darkness in a grayscale. Total darkness is given the value of 0.

HiTechnic color sensor Contrary to other sensors, this sensor is manufactured by HiTechnic. Nevertheless their products are certified by Lego. They bring industrial grade sensors to the NXT ecosystem. This sensor is able to detect colors: white, blue, red, etc.

Ultrasonic sensor Uses the speed of sound in the air to measure the distant to an object in front of it. Is able to measure distances from 0 to 2.5 meters with a precision of $\pm 3cm$. Flat solid objects provide better distance estimates than curve or less dense ones. Two or more sensors in the same environment can interfere with each other.

Servo motors Provides the torque and power necessary to move the contraptions designed with the kit. It provides rotational feedback to synchronize the speed between motors. Rotations can be accurate to $\pm 1°$. A built-in gear reduction is included with the motor.

Lamps Generates light. Is possible to turn them on or off.

2.1.2 LEGO MINDSTORMS Education NXT

A software development environment comes bundled with Lego Mindstorms called LEGO MINDSTORMS Education NXT. It was developed in conjuction with National Instruments, the company who develops LabVIEW.

LEGO MINDSTORMS Education NXT provides component system that is operated through a graphical environment. It follows the dataflow programming paradigm which focuses on how components are connected. Physical lego items like sensors, servo motors and abstract logical conditionals or cycles have their software component representations. The users build applications by dragging components from a palette and dropping them in a working area.

2.1.3 Open source nature

Lego Mindstorms provides all the necessary software and hardware to start working with the robotics kit out of the box. Also, Lego released a complete documentation including hardware design, schematics and sdk documentation for the whole platform. This encouraged the growth of a community of hobbyists and professionals who have extended the platform.[31]

It is possible to program a NXT device in development environments based on languages like C++, C, C, Java, Matlab, Lua, Ada Ruby, Phython, etc. Third party sensors like compasses, gyroscopes or even GPS have been integrated.

2.2 Bluetooth

2.2.1 Introduction

Bluetooth was created by the telecommunication manufacturer Ericsson in 1994 as a means to provide wireless communication over short distances between electronic devices. It uses the unlicensed frequency of 2.4 MHz and the transmission method of frequency hopping spread spectrum which uses the electromagnetic spectrum in an efficient way. [34]

This network of wireless devices, also called piconet is composed of master and slave devices. Master devices control how the communication is performed and can communicate with up to 7 slave devices.

2.2.2 Pairing process

A security mechanism called pairing was implemented in Bluetooth to control the exchange of data between devices. The pairing process starts with an initial request sent from one device to another and a secret code only known by the two devices being paired. When the pairing request arrives to the second device user

interaction is needed to enter the secret code. If the secret code is correct both devices become paired. Future connections won't need user interaction.

2.2.3 Profiles

Bluetooth devices can vary greatly on their intended capabilities, some of them are designed transmit audio others video, files, faxes, etc. The Bluetooth specification defines the required services, features and behavior for each capability which is called profile. Profiles are implemented on top of a core specification. Bluetooth devices can discover what kind of profiles are available for each connection.

2.2.4 Serial Port Profile SSP

This profile implements the RFCOMM protocol, which emulates RS-232 serial cable communication. It provides two-way reliable serial data communication between bluetooth devices. A wireless version of RS-232 was the original goal when Bluetooth was originally designed.

2.2.5 Bluetooth in Lego MINDSTORMS

Lego Mindstorms support Bluetooth communication with Serial Port Profile with certain limitations.

1. A NXT device supports up to 3 slave connections. Slave devices are initialized first and wait for a Master device to start the connection process. The architecture of Bluetoothh piconet of NXT devices is shown in figure 2.1.

2. The NXT device can only communicate with one device simultaneously.

3. Bluetooth SSP provides reliable communication. Nevertheless the implementation in Lego Minstorms NXT is not. The reason for this is the architecture chosen. Before data is transmitted to another device, is placed in a buffer. When the buffer is full, old data is deleted. This means we have no guarantee that data will arrive to its final destination. A solution to mitigate this issue will be discussed in latter chapters.[32] [38]

2.3 LeJOS

2.3.1 Introduction to LeJOS

One popular open source project based on Java to develop applications for NXT devices is LeJOS. It replaces the default NXT firmware with a custom one that implements a restricted Java Virtual Machine. Contrary to the default drag and drop environment that is provided out of the box with Lego Mindstorms, leJOS targets advanced users that want to benefit from object oriented programming. [39]

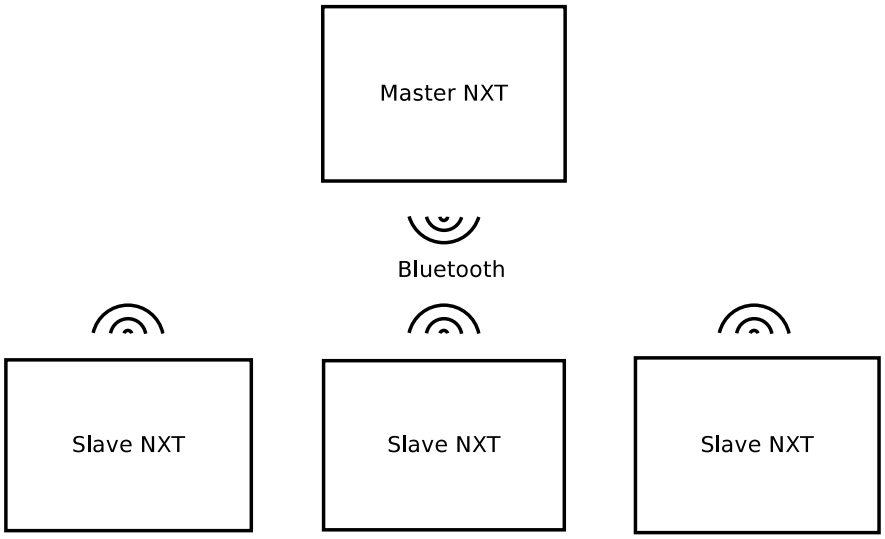


Figure 2.1: NXT Bluetooth communication architecture.

LeJOS ports the feature rich JVM in a constrained environment like a NXT device. Although Java officially endorses JME as the Java platform of choice in embedded devices LeJOS does not attempt to comply 100% with it with it neither with JSE. It takes features from both platforms with minimal changes. Hence the standard bootstrap classes of JME or JSE can not be used. LeJOS provides a custom set of bootstrap classes bundled in a classes.jar file.

2.3.2 LeJOS API

Two different API are provided with LeJOS. The fist one is used to create applications that will run exclusively on the NXT. It is used in environments where the NXT is expected to work independently of an external device like a PC. The second API called PC API was designed for applications that go beyond the computing power of the NXT for example digital image processing, when the full JSE features are required like reflection or when the NXT is intended to be remotely controlled. It makes possible the creation of LeJOS applications that will run partially on a standard JSE JVM and partially on a NXT device.

The PC API supports the remote execution of commands using a protocol specified by Lego called LEGO MINDSTORMS NXT Communications Protocol (LCP). It defines a set of commands triggered by Bluetooth messages that control directly the hardware of the NXT.

Both API define abstractions for all Lego Mindstorms components and third party sensors. Also higher level abstractions like PID controllers or navigation algorithms are implemented.

2.3.3 LeJOS features

LeJOS partially implements JSE and JME, it offers a multithreading virtual machine and Java 6 language features like inheritance, interfaces, generics and instanceof keyword. The virtual machine also implements a garbage collector.

Various LeJOS tools are provided: Eclipse and Neatbean plugins ease the development process and command line executbles to flash the NXT firmware, compile, link upload and run applications.

2.4 JME

Sun Microsystems developed Java with the "Write once, run anywhere" motto. The platform was designed for the Internet and commercial programming and JSE was created as the core platform, enterprise support was offered with a bigger platform called J2EE. Nevertheless people started to create Java applications for embedded devices and smart cards which were not the original target. Sun decided to define a set of standards to bring the Java platform to these constrained environments through JME. [13]

Sun realized that the embedded devices range was to wide for a "one size fits all" solution and decided build the JME specification around configurations, each configuration defines the hardware constraints for a small range of devices:

- Memory configuration
- Processor architecture and clock speed
- Communication hardware available

JME specifies two kinds of configurations:

2.4.1 Connected Limited Device Configuration - CLDC

CLDC is targeted to the low end range of embedded devices with memory around 512 KB. In this range fall low end cell phones and PDA devices. The applications for this configuration are called Midlets, which users can download and install wirelessly to the device. Some features from JSE are dropped for this configuration:

- Lack of the interface `java.io.Serializable`. This configuration was not designed to support distributed applications hence features like serialization or RMI are not supported.
- Most reflection features are dropped specially the classes and interfaces in the package `java.lang.reflect` are not present. Also methods were removed in the class `java.lang.Class`.
- No support for custom class loaders
- No support for deamon threads or groups.

2.4.2 Connected Device Configuration - CDC

This configuration is targeted at high end embedded devices, typically 2MB of memory. Average devices include high end cell phones or set-top boxes like blu-ray. Optional packages include support for JAVA SE RMI to deploy distributed applications and JDBC 3.0 API to standardize the manipulation of relational databases.

2.5 Code optimization

The limited hardware present in embedded devices like LeJOS or JME requires careful design and implementation of software solutions to be suitable for deployment in such constrained environments. On the other hand, the source code written by a programmer focuses mainly on readability and maintainability but not necessarily on code optimization. It is possible to have a proper balance between these features thanks to the code optimization tools used in compilers. They are able to automatically reduce the size of a program.

2.5.1 Control Flow Analysis

Control Flow Analysis is a type of static code analysis which analyses how a computer program is executed. It stars by defining the *basic blocks*. A basic block is a set of continuous statements that comply with two requirements. The first requirement states that a basic block can not have jumps or branches i.e statements with conditionals or GOTO functions. The second requirement is that only the last statement can transfer the execution to a different block. To perform the analysis a control flow graph is created whose nodes are the basic blocks identified and the edges the relationship between them according to the program flow. Using this graphs loops can identified and the flow control analyzed. [11]

2.5.2 Data Flow Analysis

Data Flow Analysis is also a type of static code that analyses how data is manipulated through the program flow. It requires the basic blocks and the flow control graph obtained at the flow control analysis phase. The analysis is performed by analyzing the effect of each building block on the variables and how a change in the data is propagated. Dead code (which means code that if removed does not affect the program behavior) can be found using this technique.

2.5.3 ProGuard

Different tools that perform optimization in Java exist like ProGuard. It operates on the byte code and uses control flow analysis, data flow analysis and other different techniques to optimize code. ProGuard is able to:

- Reduce unnecessary method calls, comparisons, instanceof operations,
- Remove classes, methods and fields which are not used not used

- Delete duplicated code
- Changes method signatures to final, static and private and class signatures to final and static if it is possible.

2.6 SOFA 2

2.6.1 Introduction

Component based software development promotes best practices like separation of concerns, implementation of loosely coupled independent components and reusability. Various component models exists today: Corba Component Model by Object Management Group, COM+ model by Microsoft, EJB (Enterprise Java Beans) model by Sun Microsystems, Fractal component model by INRIA and France Telecom.

The Department of Distributed Systems and Dependable Systems at Charles University designed and implemented an innovative component model called SOFA 2. This model is the result of their research and experience developing component oriented applications.

2.6.2 Component model

SOFA 2 applications are organized in a hierarchy of components. Two types of components are defined in SOFA 2: primitive components and composite components. Primitive components cannot have child components (subcomponents), they are implemented by providing concrete business logic. Composite components on the other hand are implemented by providing references to subcomponents, composite components do not implement business logic on their own. Figure 2.2 shows an example with both types of components.

SOFA 2 components take the object oriented abstractions of interfaces and classes to a higher level. It defines Frames and Architectures. Frames are the analogy of interfaces, they provide a black-box view of a component and define which interfaces are provided or required. Architectures are concrete Frame implementations, they specify the subcomponents and how they are connected, only one level of nesting is specified for each component.

SOFA 2 defines a meta-model to represent the whole component system. This model has an Architecture Description Language ADL implemented in XML which is used to define the interfaces, frames, architectures, subcomponents, how subcomponents are connected, which is the communication style used between components, where the components should be deployed, etc.

The component model defines connectors, these are used model the binding between components on the same level of nesting. They also specify how components will interact and they are responsible for seamless distribution of components across devices.

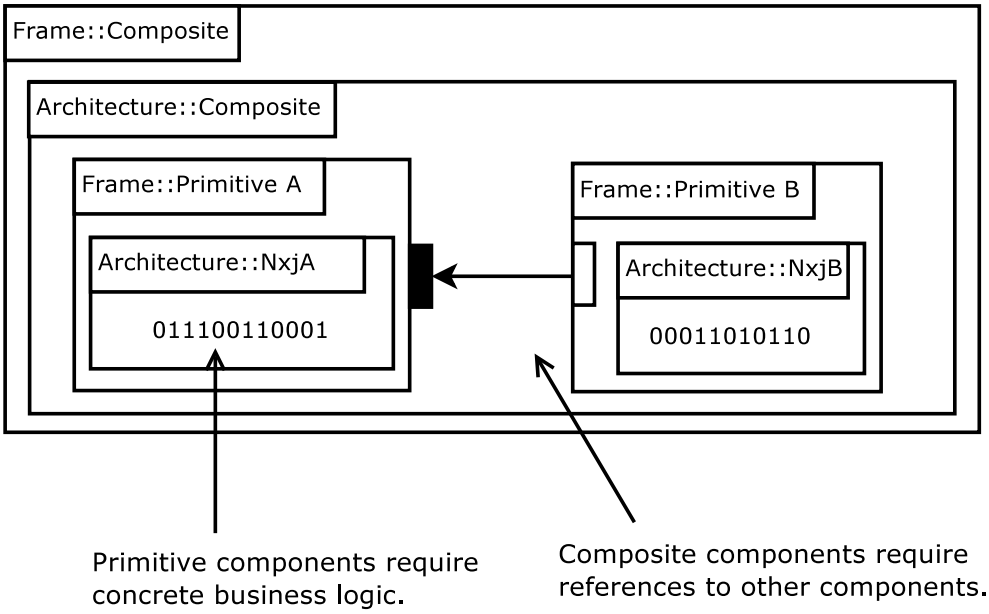


Figure 2.2: Diagram of the SOFA 2 system that shows primitive and composite components.

2.6.3 Dynamic reconfiguration

Dynamic reconfiguration in SOFA 2 [1] permits the modification of the application's architecture at runtime. This includes the addition and deletion of components and management of the references to components. SOFA 2 solves this problem by supporting controlled reconfiguration. This alternative is more desirable than to forbid dynamic reconfiguration or to authorize uncontrolled reconfiguration.

2.6.4 Controller interfaces

Components in SOFA 2 also include support for non-functional properties through control interfaces. This means that by using these interfaces components: *a*) Components became dependent on the environment they are running *b*) Components can benefit from information supplied by the environment. Experience gained trough the development of non trivial component application showed that is desirable for components to be aware of their environment [1].

The smallest elements in the Controllers are the microcomponents. These are called micro because there is no hierarchy (all are primitive) there are no connectors between them, they are not distributed and do not have a control part. They are just classes that implement a defined interface.

The next element in the control hierarchy are Aspects. They are composed of microcomponents. The control part of a SOFA 2 component is conformed by a set of Aspects and they are specified at deployment time.

2.6.5 Runtime environment

SOFA 2 not only provides development tools but also a runtime environment where components are deployed called Dock. SOFA 2 applications can be distributed among Docks which belong to a SOFAnode. The location where components must be deployed is defined using a deployment plan.

Docks are responsible for initializing the components, initializing component's aspects, initializing the connectors and starting the application.

For a Dock to became operational in a SOFAnode it has to be registered in the Dock Registry. The Dock Registry provides a lookup mechanism to discover the Docks that belong to a SOFAnode. At deployment time, the Dock Registry is queried by the Dock that starts the deployment and gets a reference to the needed Docks.

2.6.6 Repository

One of the stated benefits of component based software development is code reuse. To help achieve this goal a Repository was implemented. It stores information about interfaces, frames, architectures, generated connectors, assembly and deployment.

Developers working with SOFA 2 use the Repository like a revision control tool. They start by defining a workspace and perform a checkout of the components from the Repository. They work on the local files, make changes or create new components. The development lifecycle finishes when the changes in the local workspace are committed to the SOFA 2 Repository.

The Repository is also used at deployment time. Docks download the components that are needed from the Repository according to the deployment plan.

SOFA 2 internally tags all classes with versions, this permits the deployment two versions of the same class in the same application without any conflict.

2.6.7 Conector generator

One benefit of SOFA 2 applications is that components do not have dependencies with a specific middleware. In the design phase, ADL is used to specify which communication style will be used for a given Frame i.e. method invocation, messaging, streaming, etc., and how the components are connected. Components are responsible for the business logic and not for the communication between them.

At the beginning of this chapter it was explained the concepts of Delegation and Subsumption and also how they fit in the connector model in SOFA 2. Those connectors are defined at design time. SOFA 2 defines also runtime connectors. The connector generator (*congen*) is used to generate runtime connectors.

The connector generator is responsible for enabling the communication between components with the communication style specified. A generalized connector model was proposed and implemented in [4]. The connector generator uses the knowledge of the communication styles needed to pick the middleware that best suits the need of an application using an optimization algorithm and implements the runtime connectors at deployment time. The benefit of creating the runtime connectors at a late state is that there is more information about the deployment environment.

2.6.8 Global Connector Manager

Following the same patter between Docks and the Dock Registry, connectors must register to the Global Connector manager. It is responsible for the connectors that span multiple Docks and its default implementation depends on Java RMI [2].

2.6.9 Cushion

SOFA 2 not only provides a component model and runtime environment but also development tools. Cushion is one of them and glues together all the tasks needed to develop a SOFA 2 application. It is implemented as a command-line tool. With this tool is possible to manage the components in the repository, assemble and deploy applications.

2.6.10 SOFA 2 JME

SOFA 2 model and its abstractions are language agnostic, Java SE is used to implement primitive components due to its advanced features like dynamic-class-loading and reflection [1]. SOFA 2 also support applications that run in embedded devices using JME. It provides a runtime environment for it and special cushion tasks[3].

JME has specific requirements which pose limitations on the kind of SOFA 2 applications that can be deployed. JME CLDC does not support a communication middleware such as RMI or JMS. SOFA 2 implementation uses this technologies in core areas i.e Global Connector Manager. JME CLDC do not support custom class loaders and its security model does not let additional code to be downloaded and used at runtime. The default SOFA 2 deployment process in JSE involves a download process of the components from the repository.

A proposed solution to overcome this limitations is defined at [3]. It consists of changes in the development process used in JSE. Since JME can not download code at runtime, a SOFA 2 application in JME is treated as monolitical. All the components will be prepared, compiled and assembled in a package that can be uploaded to a JME device. Also software developers must be aware that the implementation of primitive components, business interfaces and micro components have to comply with the requirements of JME. A consequence of the lack of middleware software is that applications can not be distributed.

3. Proposed design for SOFA 2 support in Lego Mindstorms

This chapter describes the proposed design to fulfill the goal of extending SOFA 2 to support the creation of distributed applications that target NXT devices. It analyses the alternatives, benefits and disadvantages. It starts by proposing a solution to support SOFA 2 runtime on the device. Then a solution is presented for the lack of middleware infrastructure in LeJOS. Finally it describes a proposal to support remote connectors in LeJOS.

3.1 SOFA 2 runtime support in LeJOS

3.1.1 LeJOS limitations

The LeJOS virtual machine does not intend to be 100% compliant with either JSE or JME but the differences with those platforms are not extensive. SOFA 2 support for these two Java platforms paves most of the way for a LeJOS implementation. SOFA 2 for JME was chosen as starting point because it shares most of the characteristics and restrictions like:

1. Lack of RMI or JMS middleware.
2. No custom class loader support.
3. Inability to download and run code during runtime.
4. Limited hardware environment.

The following list contains additional limitations unique to the LeJOS environment:

1. Partial implementation of Hashtable class: This class only supports methods to insert, retrieve by key and list all keys. It does not provide a remove method, a method to retrieve the elements, a method to assert whether the hashtable is empty and a method to assert whether a key is present in the hashtable.
2. Partial implementation of Random class: method `setSeed()` is not implemented.
3. Some methods in the `Class` class can not be used: LeJOS virtual machine does not implement reflection or classloader support. Some method signatures in this class are only provided to be able to use the standard Java compiler (javac) or the Jikes compiler. A Call to any of these methods will end with an Exception.
4. Partial implementation of JME: LeJOS does not define Midlet applications and does not follow any program execution lifecycle.

3.1.2 Analysis of LeJOS runtime environment and requirements

LeJOS shares most of the restrictions analyzed at [3]:

1. A NXT device like JME, is not capable of running a SOFA 2 deployment dock nor be part of the SOFANode infrastructure.
2. Like JME, the code available at runtime is restricted to the code supplied at link time. No additional code can be added at runtime.
3. LeJOS like JME also uses custom bootstrap classes which differ from JSE.
4. NXT devices provide a communication mechanism that can be used to support remote method invocation, in contrast to JME which is restricted to local method invocation.

LeJOS and JME share most of the restrictions that have a direct impact the implementation of a SOFA 2 runtime environment. Thus the proposed changes in the software development process described at [3] to support SOFA 2 in JME can also be applied to SOFA 2 for LeJOS. Thus, the current implementation of SOFA 2 for JME can be easily extended to produce a monolithic SOFA 2 for LeJOS application that runs in a single NXT device. The significant changes are located on the connectors between NXT devices to support the seamless distribution of components, a feature which is currently not available on SOFA 2 for JME.

3.1.3 SOFA 2 runtime in LeJOS

In this section we analyze the differences between LeJOS and JME and how this differences impact the development of a LeJOS runtime environment for SOFA 2.

Hashtable class

SOFA 2 for JME runtime depends on the class Hashtable class, whose implementation in LeJOS is incomplete. The alternatives analyzed to find a solution for the Hashtable limitation are: *a*) Implement a new Hashtable with all the required functionality, *b*) Extend Hash table and implement the missing functionality and *c*) Use the current implementation and create a helper class that implements the missing functionality by calling static methods.

Solution *a* has the disadvantage of partially re implement functionality which works well in LeJOS. A disadvantage of solution *b* is that can depend on the internal implementation of LeJOS Hashtable hence less maintainable in future versions. Solution *c* keeps the design simple because there is only one class that implements a hashtable, thus it offers a unified way to work with hashtables. Also this solution has less coupling than *b* and more code reuse than *a*. Solution *c* was thus preferred over *a* or *b*.

This strategy will be applied for all cases that require to extend the functionality of LeJOS API due to lack of compliance with either JSE or JME.

Random class

One of the sample SOFA 2 component applications that was ported to LeJOS was *logdemo*. This is a simple application which uses two components and logs messages to the screen. In this thesis it was used to test the SOFA 2 runtime environment for LeJOS.

The current implementation of *logdemo* uses the method `setSeed()` method of the `Random` class. This method do not exist in the LeJOS `Random` implementation. Furthermore in the current implementation of *logdemo* the seed is set to a constant value, this means that all the executions of this application will generate the same sequence of random numbers. In the LeJOS implementation of *logdemo* the system time will be used as the seed to generate random numbers.

Class class

Method `toString()` in of the class `Class` in LeJOS has different semantics compared to JSE. In LeJOS each class is mapped at link time to a unique number identification. A call to the `toString()` method will retrieve such number which can useful to debug a LeJOS application.

Development process

LeJOS shares most of the restrictions analyzed in JME applications. The development process defined in [3] in the Chapter 3.2 can be reused with minor changes:

1. Instead of compiling agains Java ME bootstrap, the LeJOS bootstrap is used. Instead of JME preverification process, the LeJOS linker is used to package the compiled byte-code into a binary code that can be executed by the custom firmware of LeJOS.

2. SOFA 2 for JME is restricted to a single Dock. Distributed deployment plans are forbidden due to the lack of RMI features of the JME CLDC configuration. A distributed deployment plan must be supported to comply with the goal of this thesis. Thus, this restriction is lifted in SOFA 2 for LeJOS.

3.2 Middleware in LeJOS

One of the most important goals in this master thesis is the support for a distributed deployment plan. In SOFA 2 for JSE RMI is used to perform remote method invocation and create connectors that span through multiple Docks. LeJOS does not provide Middleware infrastructure [1]. Although LeJOS has a vibrant community, no third party library was found to implement RMI or any other Middleware. Hence part of the effort in this thesis was used to design and implement a light

[1] In the official LeJOS roadmap `http://sourceforge.net/apps/mediawiki/lejos/index.php?title=Roadmap:Future_Projects` is stated that RMI support is a future task.

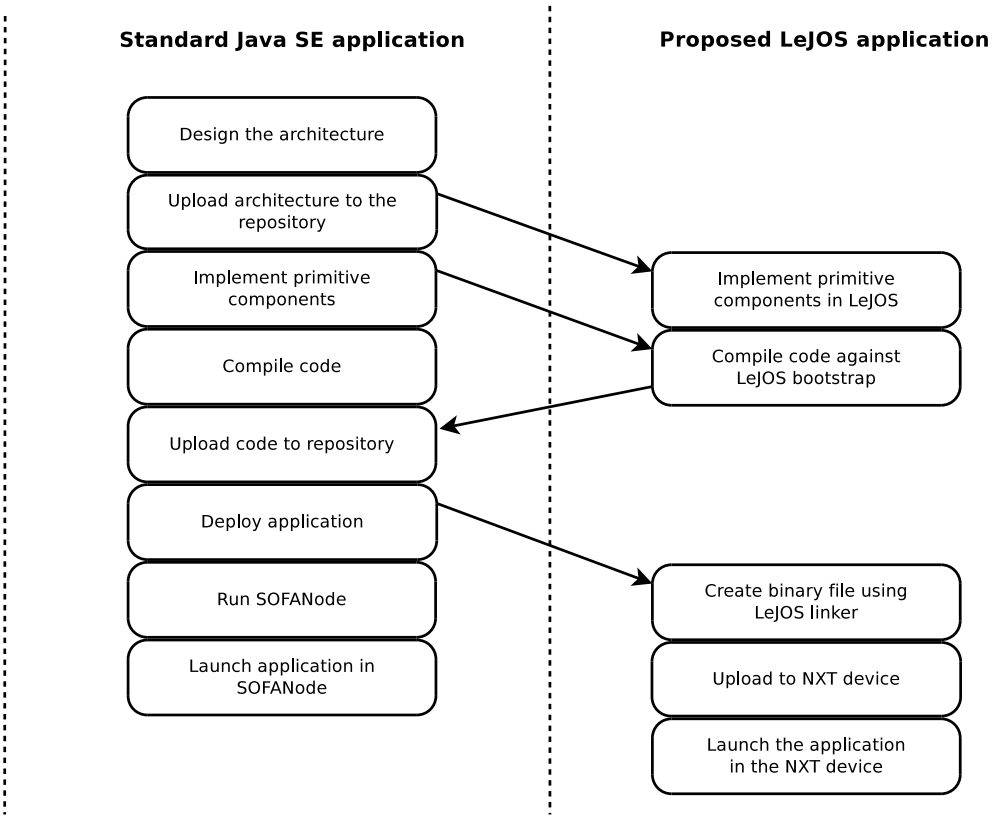


Figure 3.1: Deployment process of SOFA 2 applications in LeJOS. Adapted from [3]

RMI Middleware to be used by SOFA 2 for LeJOS.

The proposed RMI is Light because it lack features with respect to its JSE counter part. The RMI in LeJOS does not implement a distributed registry, instead each device will control the resources it provides. Also the types of the parameters or responses of a method invocation are limited to primitives, Wrapper classes or Strings.

The middleware software was designed as a layered architecture, which has proven to be a good design in communication systems. By providing reliable communication, a messaging layer is built on top of it. At the higher level a RMI layer was implemented to provide support for SOFA 2 required and provided interfaces.

3.2.1 Reliable communication

Design

We showed in Chapter 2.2.5 that LeJOS API provides access the NXT Bluetooth hardware and that Bluetooth communication between two NXT devices is not reliable when 3 or more devices are connected. Bluetooth with SPP is itself reliable but the hardware architecture chosen by Lego is not.

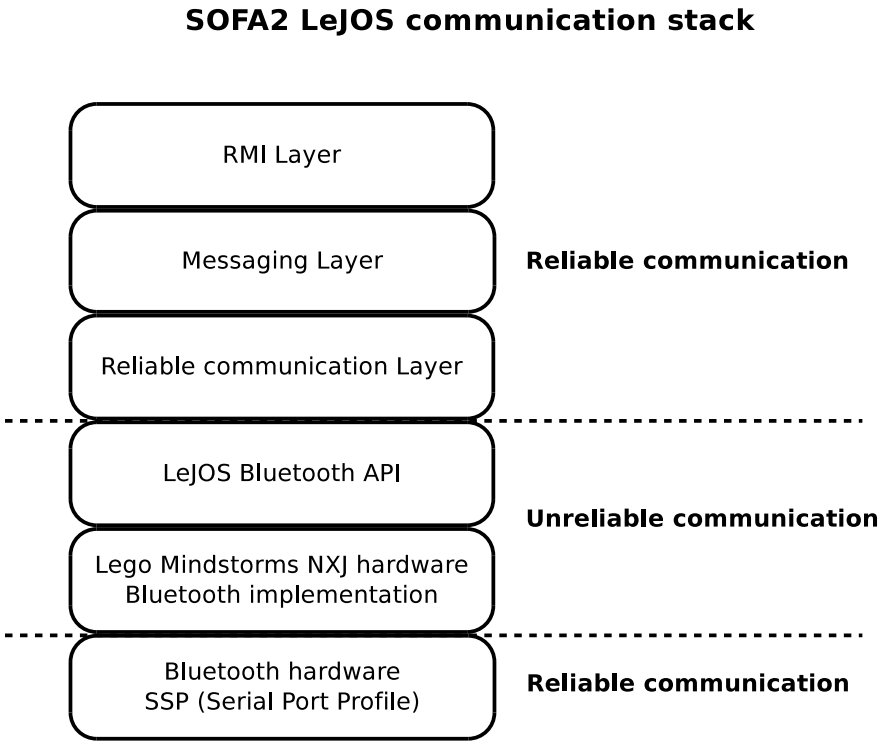


Figure 3.2: Layered architecture of the Bluetooth communication in SOFA 2 for LeJOS.

Two alternatives were analyzed to make the communication reliable. *a*) implement a mechanism of flow control and congestion control similar to the *Transmission Control Protocol* which uses a *sliding window protocol b*) a simple flow control mechanism in which a slave NXT device will transmit data only when it is requested the master NXT device.

Alternative *a* uses the link channel efficiently. Alternative *b* guarantees that only one slave device will send data to a master device thus no data will accumulate in output buffers. Alternative *b* is less complex to implement compared to alternative *a* thus better suited for an embedded device.

An algorithm was designed that works in a similar way to a *token ring*. A token is sent back and forth between the master device and the slaves connected to it. The master selects the next slave based on a round robing principle. Only a device in possession of a token is pemitted to transmit data to another device.

3.2.2 Message layer

The main requirement for this SOFA 2 LeJOS middleware is to support connectors that are able span multiple Docks. The communication system at the reliable layer provides an independent set of reliable links between the master and slave devices. An infrastructure to coordinate these links is required. Following the same pattern used in communication systems, an address is assigned to each device. This address corresponds to its Bluetooth name. At deployment time this same address will identify the Dock to which components belong. This address

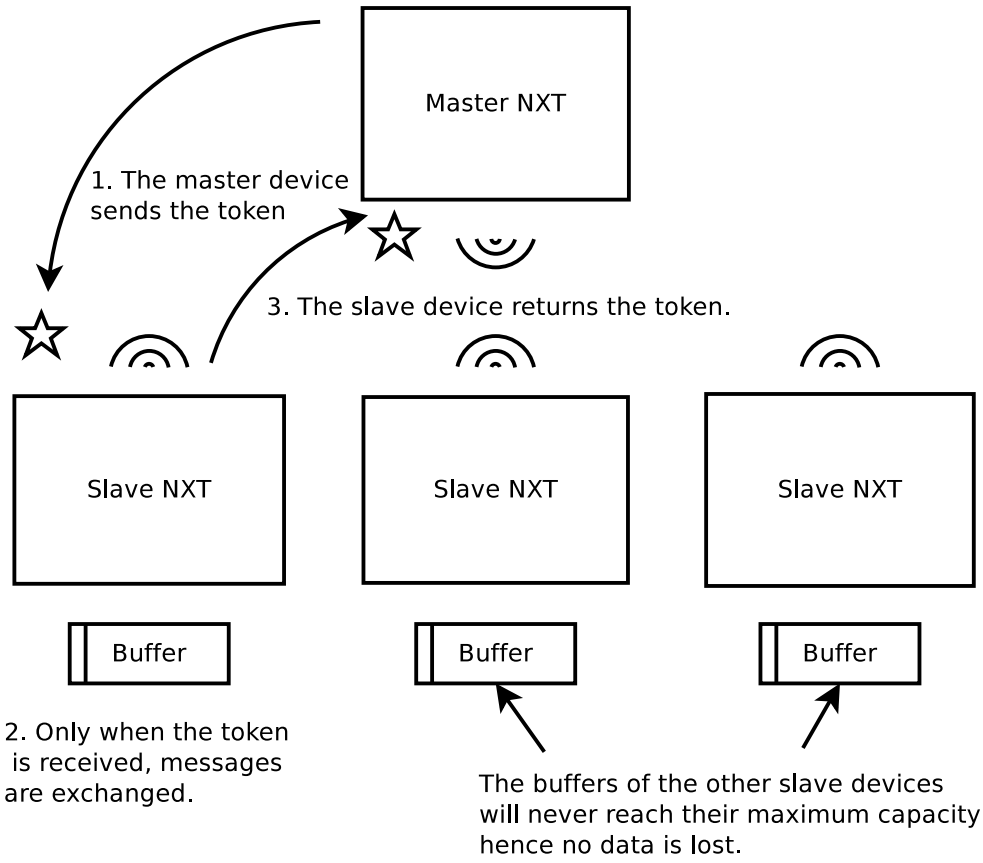


Figure 3.3: Token ring implementation to create a reliable communication between devices.

must be unique per piconet. In this thesis docks are named "nodeA", "nodeB" and "nodeC". There will be a NXT device per each Dock and the Dock "nodeA" plays the role of the master device that initiates the Bluetooth connection.

Synchronous vs Asynchronous

An important feature to be discussed for this message layer is whether it should support synchronous or asynchronous communication. These two alternatives were analyzed. Asynchronous solutions have the benefit of being more decoupled due to the fact that the request and response are not tightly bound. To find the best alternative a clear understanding of how hardware Bluetooth operates on a NXT device is needed. The following facts are important: *a*) In the section 2.2.5 it was explained that a NXT device can only communicate with another device at the same time. *b*) The Bluetooth hardware takes up to 250 ms to switch between one device to another, [2] *c*) There are no direct communication channels between slave devices. All the messages between two different slave nodes must be served through the master device.

In other network technologies like Local Area Networks (IEEE 802) any mem-

[2] Further explanation of the Bluetooth features in LeJOS are explained at `http://lejos.sourceforge.net/forum/viewtopic.php?t=1964`

ber of the network can communicate with other peers at any time [3], by contrast LeJOS devices must wait for their turn to be able to transmit any message. If a given turn is excessively long it will affect the communication process of other devices, thus an strategy has to be found to make fair usage of the communication channel. The proposed solution is to use asynchronous communication and support blocking operations as close as possible to the application layer. It is possible to take advantage of the reliable layer to simplify the asynchronous communication process by skipping the acknowledge of messages.

Message nodes

In the middleware design all devices are treated as message nodes whether they are a master or a slave. The message node abstraction provides the mechanism responsible for sending and receiving messages. This abstraction is also used to define the physical communication architecture of the pico net, by registering the devices that communicate with a Message node.

Message format

At the message layer, messages are void of any semantics. Messages at this layer are only used as a mechanism to transport information from a device to another.

A message format was defined for each layer in the middleware. At the reliable layer a flag is used to send back the token to the master device. At the message layer, the format contains the information about the sender, receiver and content. The master device is responsible for routing the message to the required receiver.

The parameters in the different layers are always on the left side of the message in such a way that each layer can retrieve easily the information it needs and pass the rest of the message to the next layer.

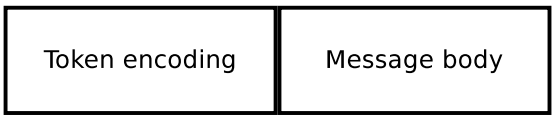


Figure 3.4: Diagram of a message in the reliable layer.

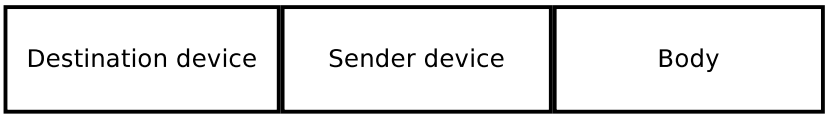


Figure 3.5: Diagram of a message in the messaging layer.

[3]Packages can be transmitted as long as the communication channel is free and no collisions are detected.

3.2.3 RMI layer

In the previous sections it is described how to transmit information between any two NXT devices. Focus is placed now on bringing support for provided and required interfaces which are fundamental items in SOFA 2 components. SOFA 2 supports different communication styles including synchronous method invocation, asynchronous messaging, distributed shared memory, etc. The semantics of a synchronous RMI are closer to a method invocation in the same address space, for this reason this communication style was chosen. This fact is further exploited in the proposed design for SOFA 2 connectors.

Java RMI implementation was used as inspiration facilitate the integration of the LeJOS middleware with the connector generation tool which has support for JSE RMI. The LeJOS RMI is a light weight version. It has the following limitations:

1. There is only support for Java primitives and their class Wrappers. There is no support for arrays or collections. This constraint does not limit the usefulness of applications that can be developed for the target device. Using only primitive parameters a NTX device can control the sensors or behavior of another device.

2. Remote interfaces can not be added at runtime. LeJOS do not support the loading of classes at runtime this means that all the local and remote interfaces need to be present when LeJOS binary is linked.

3. No support for a Registry. A service to get references of remote objects do not exist, remote devices need to know on advance of the existence of remote objects. In JSE a distributed service exists to which remote objects register and clients query to obtain remote references. This restriction does not limit the SOFA 2 applications that can be developed for LeJOS since it is known in advance from the deployment plan the required connections between the components.

Features

The RMI design follows the traditional stub & skeleton design as depicted in the Figure 3.6. Instead of having a centralized Registry, each Message node has a local Registry that maps a remote object with a concrete Object server.

The LeJOS language supports interfaces, thus a SOFA 2 interface used to describe required or provided interfaces can be also supported. Interfaces will be used to work with the generated `Stubs`.

JSE comes with a RMI compiler called `rmic` which is used to generate the stubs & skeletons. A similar tool was designed but instead of generating byte code, it generates Java code. This facilitates the processes of implementation and debugging.

In a similar way to CORBA interoperable object reference (IOR) the LeJOS Middleware was designed with one simple string identifier. A unique identifier for the whole piconet has to be used, again this imposes no restriction because all the information needed for the connectors is given in the deployment plan. This unique identifier is used to create the Stubs and can be used from anywhere in the pico net.

When a method invocation is performed on a Stub, the local Repository is queried to determine if the Message node is serves the resource. If it is the case the call is not marshalled/unmarchalled, local method invocation is used directly. This feature will permit further simplifications in the Connectors which will be explained in the next section. If the request is not served in the current Message node, the request is tagged with a unique identifier per Message node, this is used to match requests and responses, since the message layer is asynchronous it might happen that a requests are not received in the same order in which they were sent.

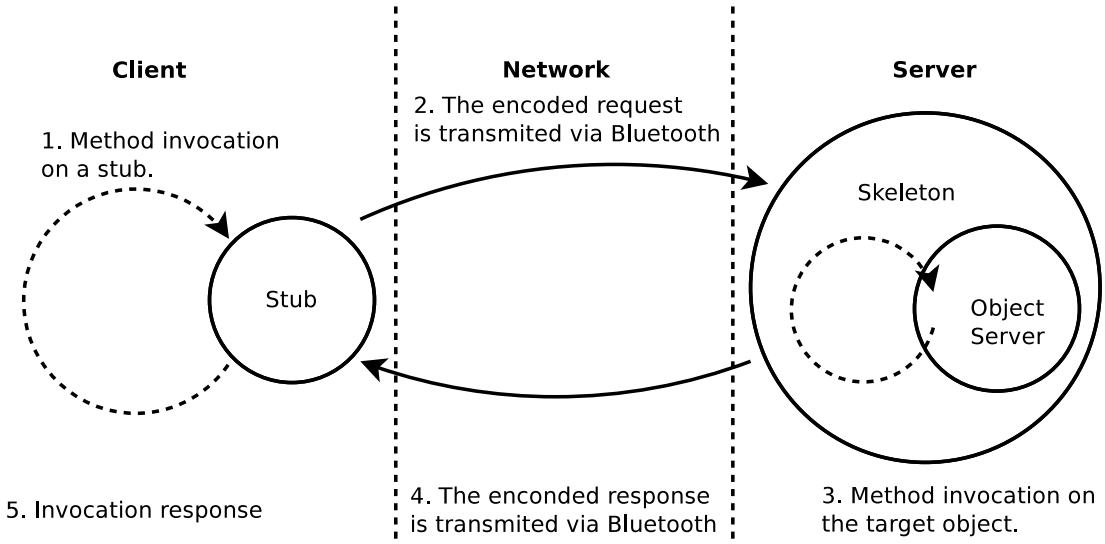


Figure 3.6: RMI lifecycle in SOFA 2 for LeJOS. [26]

3.3 Connectors in LeJOS

The SOFA 2 runtime for JME proposed by [3] provides an implementation that support local connectors using with the communication style of *method invocation*. LeJOS shares most of the restrictions and features of JME with CLCD profile. Thus it is possible to extend work done by [3] to support local connectors in LeJOS. An important goal of this thesis is to support connectors that span multiple Docks, a feature which is no possible in the current SOFA 2 for JME. This section analyses the possible solutions to this problem.

3.3.1 Extend the congen tool

SOFA 2 depends on the *congen* [4] tool for the automated generation of connector code in SOFA 2. These connectors are responsible for the communication

between components whether they reside in the same address space or not. The *congen* design and implementation defines a set of code templates that simplify the support of new middleware. New templates can be created to support the LeJOS RMI based on the existing JSE RMI.

A disadvantage of this solution is that some of the implementation classes in *congen* like `GlobalConnectorManager` depend on RMI and define remote object interfaces which are forbidden in the Light LeJOS RMI. Most of the objects used to configure the connectors could not be serialized and sent to a remote NXT device.

SOFA 2 and *congen* were designed to be fully dynamic and use all the rich features provided in JSE but with most of this advanced features unavailable in LeJOS, further alternatives need to be analyzed to support distributed connectors in this constrained environment.

3.3.2 A new connector generator tool for LeJOS

A completely different approach is to create a special connector generator for LeJOS. A NXT device running the LeJOS custom firmware has a different computing environment from JSE and JME. A new *congen* tool could be designed and implemented for this device.

The main advantage of this solution is that a tool can be designed and implemented to fulfill the specifications, features and restrictions of distributed connectors in SOFA 2 for LeJOS.

A huge disadvantage of this alternative is that designing and implementing such connector generator from scratch would fall beyond the scope of this thesis. On the other hand it would would require to design an implement features which are already present in *congen*.

An optimal solution needs to extend or reuse *congen* as much as possible.

3.3.3 Modified congen

The last alternative lies on the middle of the previous solutions. It reuses as much as possible *congen* and satisfies the restrictions of LeJOS, it consist of:

1. Maintain the connector architecture intact as proposed in *congen* i.e. the structure of the classes `ClientUnit`, `ServerUnit`, `LocalSleleton`, `LocalStub`, etc. is preserved.

2. When a SOFA 2 deployment plan is committed to the repository, *congen* will generate standard JSE RMI connectors as it occurs in its current implementation. Nevertheless those connectors won't be deployed to a LeJOS application.

3. Propose a new cushion NXJ action to generate the connectors based on the limitations and features of LeJOS. The dependencies to the JSE RMI classes will be removed and support for the Light RMI will be added.

4. The LeJOS connector generation will be delayed as much as possible, the application can have access to the maximum of information available to generate the connectors.

5. The connector architecture in LeJOS is further simplified to only support `LocalSkeleton` and `LocalStub` even when the connectors are split in different address spaces.

6. The operation and implementetion of *congen* tool is not modified.

7. The Middleware designed for LeJOS support messages and synchronous RMI. Both communication styles can be supported by this design, only the method invocation was implemented.

Since *congen* continues to operate in the same way, SOFA 2 frames and architectures developed in JSE which are compatible with LeJOS can be reused without changes. This solution does not change the development process or functionality of other SOFA 2 components developed in Java or JME.

The simplification of the connector architecture to only permit only one type of connector element either `LocalSkeleton` or `LocalStub` for both local and connectors that span multiple Docks does not impose any performance penalty. This is due to the fact that LeJOS middleware is optimized to perform direct method invocation when it is available instead of routing the request through the middleware infrastructure.

The main difference between the modified congen and 3.3.1 is that in this alternative the business logic of *congen* is not reused, only the architecture of its connectors. The connectors are not generated by *congen*, they are generated by a special Cushion task.

The main advantage of this solution is that it benefits from the connector abstractions defined in *congen* and reuse it's implementation as much as possible.

The main disadvantage of this solution is that SOFA 2 loses cohesion in the connector code generation, the logic is now split in the *congen* tool and the packages needed to deploy LeJOS components in SOFA 2. Another disadvantage is that the connectors generated by in this way will not benefit from the optimization algorithms implemented in *congen*.

3.3.4 Proposed solution

The previous alternatives were analyzed and the modified *congen* solution was identified as the best alternative. There is no need to support a full RMI solution as proposed in the section 3.3.1. On the other hand the connector generator and architecture is reused as much as possible.

4. Design implementation using LeJOS and custom middleware

This chapter describes the process of implementing the design explained in the previous chapter. It starts by showing the changes needed to support a SOFA 2 runtime in the NXT device. Then it shows how a middleware with RMI capabilities was implemented on top of LeJOS API. It follows by describing the changes needed to support remote connectors to enable distributed applications. This section ends with an analysis of the limitations of this implementation.

4.1 SOFA 2 Runtime in LeJOS

4.1.1 Runtime implementation

The support for LeJOS is implemented in the projects:

sofa-lejos-api Defines the micro components.

sofa-lejos-commons Defines the classes that are common to any LeJOS application and which depend only on LeJOS API.

sofa-lejos-runtime Defines the connectors and component architecture.

sofa-nxj-bootstrap-lejos Defines the components bootstrap classes.

The build process in those projects was modified to compile agains the LeJOS bootstrap classes instead of the JSE standard bootstrap.

Hashtable

Subsection 3.1.3 analyzes the limitations of the `Hashtable` class in LeJOS. A helper class `HashtableUtils` was created to support *isEmpty()*, *containsKey()*, and *size()* methods. For the rest of the missing methods the following alternatives were used:

remove(key) was replaced with *put(key, null)*.

elements() was replaced to a combination of *keys()* and *get(key)*.

LCDLogger

The class `LCDLogger` is used as a substitute of Java `System.out.println();` to facilitate the output of information to the LCD screen of the NXT device.

4.1.2 Cushion

SOFA 2 Cushion tool was designed to be extended easily, new functionality can be defined by `actions`. They must extend the class `InitAction` and be registered in the file `lejos-action-register.xml`. The following two actions were created.

Init Action

SOFA 2 uses a workspace folder to have a local copy of the components stored in the repository, create new ones or manipulate existing ones. The init action initializes the workspace, is implemented in the class `SofaLejosInitAction` and it configures the LeJOS environment. The environment consist of the following variables:

lejos Specifies the language of the components, use `lejos` for LeJOS. Has to be defined by the user.

bootclasspath Specifies the path to the bootstrap LeJOS class `classes.jar`. Has to be defined by the user.

classpath Specifies the path to additional resources that are needed to compile a SOFA 2 LeJOS application. Can not be modified by the user, SOFA 2 automatically sets this value.

nxjlinkpath Specifies the path to the LeJOS linker tool. Has to be defined by the user.

Both the *bootclasspath* and *nxjlinkpath* can be found in the binary distribution of LeJOS. After the *init* action is executed, a hidden file `.sofa` is created on the workspace with the supplied information.

Nxj Action

In Figure 3.1 it is shown how the LeJOS development process divert from Java SE at deployment. This change is reflected in the `nxj` action which is a modified version of the `midlet` action defined at [3]. This action is implemented in the Java class `NXJAction`, it receives as a parameter the deployment plan and it outputs binary packages that can be uploaded to a NXT device, one per each Dock defined at the plan.

The class `SOFAMidletGenerator` defined at [3] was refactored to `SOFALejos-Generator` to comply with the compile and link process in LeJOS, generate the required source code and support for deployment in more than one Dock.

4.2 LeJOS light RMI middleware

This section describes the relevant abstractions implemented on the middleware. Each layer in the communication process described in the Figure 3.2 has a Java interface counterpart, this means that interface implementations are only responsible for one layer. The classes `Message` and `MessageNode` are the exception to this rule since they cover more than one Layer.

The middleware is required to be asynchronous, this was achieved by implementing the *Observer* pattern. This means messages can be send without blocking the thread. If a class require access to the incoming messages, it must register itself to the `MessageNode`. Registered classes are notified when messages

arrive.

4.2.1 Bluetooth link and reliable communication

In this layer a single class is responsible to perform the low level Bluetooth communication, it models a single link between two Devices and handles the flow control tokens. The flow control abstraction is defined in the interface `FlowControlCommunication` and is implemented by `DeviceCommunication`. This class is composed of:

1. A reference to the Bluetooth id of the remote device.
2. A queue with the messages that must be transmitted when the token arrives.
3. A reference to indicate whether the link is inbound or outbound. Slave devices are configured as inbound and wait for a master device to initiate the Bluetooth pair process. Master devices are configured as outbound.

The flow control token was implemented as a flag with two values: 'C' means that the flow of messages continues. 'T' means that the flow terminates and the token is sent back to the master device.

```
public interface FlowControlCommunication extends TokenHandler {
  public void receive(String flowControlStatus, String message);
  public void send(String flowControlStatus, String message) throws Exception;
}
```

Listing 4.1: `FlowControlCommunication` interface.

4.2.2 Messages

The message layer introduces a `Message` abstraction which is implemented as a Java Bean to store all the information needed to exchange information between any two devices and static methods to encode and decode messages at different layers.

```
public class Message {
  private String destinationId = null; //Defines the recipient of the message.
  private String senderId = null; //Defines the sender of the message.
  private String flowControlStatus = null;
  private String body = ""; //Message's body
  public static Message parseFlowControlLayer (String text) {...}
  public static String encodeFlowControlLayer (
        String flowControlStatus, String body) {...}
  public static Message parseMessagingLayer (String text) {...}
  public static String encondeMessagingLayer (
        String destinationId, String senderId, String body) {...}
  ...
}
```

```
public interface MessageCommunication {
        public void receive (String destinationId, String senderId, String message);
        public void send (String destinationId, String senderId, String message);
}
```

Listing 4.3: `MessageCommunication` interface.

Listing 4.2: `Message` interface.

The class `MessageNode` which implements `MessageCommunication` defines the architecture of the piconet i.e. which devices will attempt an inbound connection and which others an outbound connection. It's also responsible for managing the listeners that must be notified on the arrival of a message. Nevertheless, it does not provide an implementation for the *send()* and *receive()* methods. This is delegated to the classes `MessageClient` and `MessageServer`.

`MessageClient` is used in slave devices and `MessageServer` in master devices. The main difference between these two implementation is that master devices must also route messages on behalf of slave devices. When messages arrive to `MessageServer` the final destination is check to determine if the message needs to be routed.

4.2.3 Light Remote Method Invocation

RMI runtime classes

The standard JME RMI implementation provides the `Registry` class which is responsible for register and look up of remote objects in a distributed application. In the Light RMI for LeJOS such functionality can not be provided due to the restriction that only Strings, primitives and their wrappers can be distributed. Nevertheless a local version of a registry do exist in two separate classes which implement the important interface in this layer called `RMICommunication` which is shown in Listing 4.4:

RMIClient Is responsible for keeping references to the remote objects which are registered as `Stub` classes and uses the message layer to dispatch the method invocations.

RMIServer Is the analogy of `RMIClient` for Skeletons and controls their life-cycle. It dispatches the request to the appropriate Skeleton and handles responses through the message layer.

Both classes were implemented as singletons and the messaging infrastructure must be configured before using them.

```
public interface RMICommunication {
  public void receive (
    String destinationId, String senderId, //Same as in the message layer.
```

```
    String skeletonId, //Unique identifier in the whole piconet.
    String requestId, //Unique identifier per NXT device
    String invocationStatus, //lindicates: unknown skeleton, exception, request or answer.
    String body );
  public void send( String destinationId, String senderId, String skeletonId,
    String requestId, String invocationStatus, String body );
}
```

Listing 4.4: `RMICommunication` interface.

Apart from the runtime classes which run on the NXT device, the SOFA 2 deployment tools were adjusted to work with this middleware. The class `MethodInvocationInterfaceChecker` was implemented to validate that a given interface can be used by remote objects according with the limitations in this RMI implementation. Also the generation of Stubs and Skeletons differ from the standard Java RMI. The implementation uses *Velocity* [1] templates and the Java Reflection API to generate source files instead of byte code.

This tool was used because software developers can easily inspect the structure of the Stubs and Skeletons during development, also because other classes in the SOFA 2 runtime are generated in this way like the `ComponentInstanceTemplate.vm` or `ConnectorInstanceProviderTemplate.vm`. The template `RMISkeleton.vm` generates the Skeleton and the template `RMIStub.vm` the Stub. The generated classes are named based on the interface with either the string "Stub" or "Skeleton" appended as suffix.

Marshall and unmarshall process

The marshalling/unmarshaling of the parameters of a method invocation was implemented in the following way:

1. For non primitives a boolean flag is used to determine whether they are null or not, then they are marshaled by computing their byte representation.
2. Primitives are marshaled by computing their byte representation.
3. For Strings an integer is marshalled to determine is length, then its content is appended to the request.

The same process is applied to marshall the return value of a remote invocation.

Methods and declared exceptions are encoded as integer and added to the request. A class with the same signature of JSE `RemoteException` was added to identify errors whose origin is the middleware.

[1] Velocity is a template engine that can be used to generate text files. `http://velocity.apache.org/`

Undeclared exception handling

When a LeJOS binary is created by the linker, each java class is mapped to a number identification. This mapping can differ when different packages are deployed to different NXT devices. When an undeclared exception occurs on the remote object, care must be taken to not confuse the class number identification received in the Stub and try to resolve it using the local mapping. The mapping of the remote id, where the servant is running has to be used instead.

RMI sample code

The following listings show an example of how the RMI middleware is configured and used. The piconet is composed of devices *nodeA*, *nodeB* and *nodeC*. *nodeA* is the master device.

```
1 public static void main(String[] args) throws Exception {
2   String remoteIds[] = new String[2];
3   remoteIds[0] = "nodeB";
4   remoteIds[1] = "nodeC";
5   MessageServer server = new MessageServer("nodeA", remoteIds, null);
6   MessageNode.setCurrentNode(server);
7   TestInterface target = new TestImpl();
8   Skeleton skeleton = new TestInterfaceSkeleton(target);
9   RMIServer rmiServer = RMIServer.getInstance();
10  rmiServer.registerSkeleton("nodeA/0", skeleton);
11 }
```

Listing 4.5: Sample code to initialize a remote object.

Listing 4.5 shows the code required in the server side. Lines 2-6 configure the message architecture: *nodeB* and *nodeC* are defined as slaves and the identification of server device will be *nodeA*. Line 7 instantiates a remote object/servant. In line 8 the generated Skeleton class `TestInterfaceSkeleton` is configured. Lines 9-10 register the skeleton in the local registry.

The string `nodeA/0` supplied in line 10 is the remote reference identifier to the remote object. The identifier uses the device identification `nodeA` followed by a unique servant code `0`. This code has to be unique in the piconet. Since the identifier is a string, it can be shared among devices, nevertheless the recipient of such identifier has to know on advance the type associated with the Stub.

```
1 public static void main(String[] args) {
2   MessageClient messageClient = new MessageClient("nodeB", "nodeA");
3   MessageNode.setCurrentNode(messageClient);
4   TestInterfaceStub testStub = new TestInterfaceStub();
5   RMIClient rmiClient = RMIClient.getInstance();
6   rmiClient.registerStub("nodeA/0", testStub);
7   testStub.helloWorld();
8 }
```

Listing 4.6: Sample code perform an invocation on a remote object.

Listing 4.6 shows the *nodeB* client side code. Lines 2-3 set up the message infrastructure. The current device is assigned the identification *nodeB* and it will connect to device *nodeA*. Line 4 instantiates the generated Stub class `TestInterfaceStub`. Lines 5-6 get a reference to the local registry and register the Stub. Line 7 shows the invocation of a remote method.

Notice that SOFA 2 for LeJOS offers seamless distribution of components, which means that software developers who implement SOFA 2 components focus on business logic and are not aware of the RMI communication details. All the initialization, configuration and calls to the RMI infrastructure are generated automatically by the connector generator.

4.3 Connectors

SOFA 2 for JME [3] implements applications that are deployed to a single Dock and support the *congen* [4] architecture. SOFA 2 for LeJOS extends this work to support multi Dock deployments.

4.3.1 Connector generator

The process starts by searching the interfaces used by frames in the deployment plan. Once the interfaces are identified, *velocity* templates are used to generate the code required for the connectors. The code generated is not uploaded to the cache connector repository used by *congen*. The following list describe the classes that are generated for each interface:

ServerUnit.vm Uses exactly the same code as *congen* with the exception that the constructor was modified to specify its identification in the LeJOS middleware. This identification is used to identify remote objects.

LocalSkeleton.vm Follows the same architecture of *congen*, dependencies with JSE RMI were removed and changed for the LeJOS middleware. A parameter was added to the constructor to specify its unique identification in the middleware.

RmiIface.vm Components in SOFA 2 are not aware of the middleware infrastructure used to communicate with remote objects. An adapter interface is used to couple a SOFA 2 component interface with a SOFA 2 LeJOS RMI interface i.e. methods must declare `RemoteException`.

RMIWrapper.vm It has the opposite functionality of RmiIface. This template generates an interface that wrap LeJOS RMI exceptions to be able to couple with SOFA 2 business interfaces.

LocalStub.vm Like LocalSkeleton, it follows the same architecture of *congen*, the dependencies with JSE RMI were removed and changed for the LeJOS middleware. A parameter was added to the constructor to specify its unique identification in the middleware.

ClientUnit.mv Like ServerUnit.vm, this template generates same code as *congen*, also adds the middleware identification in constructor.

Each of the interfaces are associated with an alias, this alias is defined in SOFA 2 ADL and is used in the path to store the code of these templates.

4.3.2 Connector instantiation

The class `ConnectorInstanceProvider` which was implemented in [3] was refactored to follow the changes in the component instantiation. The deployment plant is analyzed to determine all the required connectors. A data structure implemented in the classes `ConnectorIdTemplate` and `ConnectorInstanceProviderTemplate` stores the connector instantiation data.

4.4 Limitations

4.4.1 Program size

A demo Swarm application was designed to show the capabilities of LeJOS applications developed in SOFA 2. It consists of:

- A series of dark tracks in a white background.
- A set of robots, each robot is placed on its own track and the track's line is followed using the color sensor of the NXT device.
- Obstacles are placed in various places on top of the tracks.
- When an obstacle is detected by the device's sonar sensor, it signals the event to other robots, they change tracks and move in a coordinated way.

A prototype of the demo application was designed and implemented using the Light RMI but without the SOFA 2 component model to familiarize with the capabilities of the LeJOS API and how the sensors operate. With a working prototype a SOFA 2 application was designed. During the implementation, when the application reached five components as depicted in Figure 4.2. The size of the binary package exceeded maximum size which is 65KB.

A code optimization step was added to the *nxj* cushion action after the code is compiled to byte code. Proguard was used to shrink and optimize the code. Nevertheless the optimization process only shrunk the application around 2%. In the official LeJOS forum [2] was indicated that the linker used by LeJOS perform most of the optimizations present in Proguard and optimized code can be shrunk at most 10%.

Tests on binary package size showed that when a primitive component is wrapped by a composed one (which does not define business logic on its own but

[2]The discussion thread can be found at: `http://lejos.sourceforge.net/forum/viewtopic.php?f=7&t=2902`

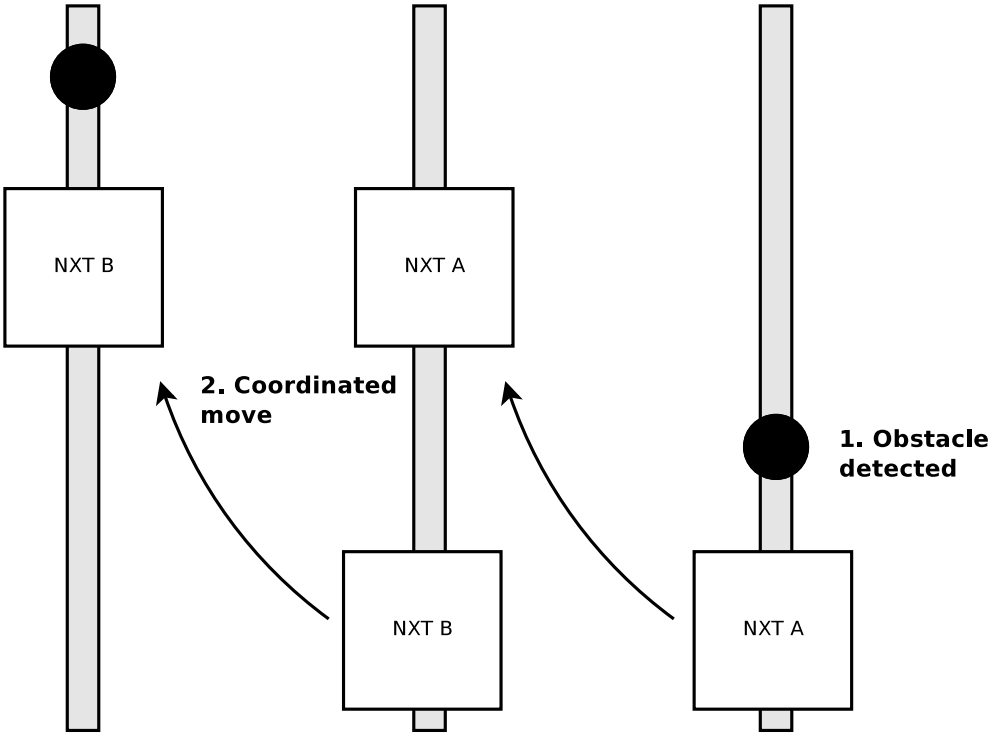


Figure 4.1: Coordinated movement in the former demo application.

only forwards the calls) there is an increase of 2KB in the package size. One of the simplest applications, log demo which was ported to LeJOS and composed of a *logger* component which prints a message to the LCD screen and a *tester* component, which calls the *logger*, has a size of 47KB.

Given the current architecture and runtime implementation, only trivial SOFA 2 LeJOS applications can be deployed or applications where the number of components is taken to the minimum. The Swarm demo application could not be implemented not run in SOFA 2, its design exceeded by far the practical limit identified of 5 components. In further sections alternatives are stated of how to overcome this issue.

4.4.2 Restricted remote interfaces

LeJOS virtual machine does not support introspection, this means that only classes which are linked to the binary package can be used at runtime. The LeJOS Light RMI middleware proposed at this thesis further restricts the remote interfaces and only permits primitives, their wrappers or Strings as part of method signatures. It would be possible to extend the middleware and support the serialization of Java Beans, arrays and collections keeping the restriction that the classes must be known at compile time.

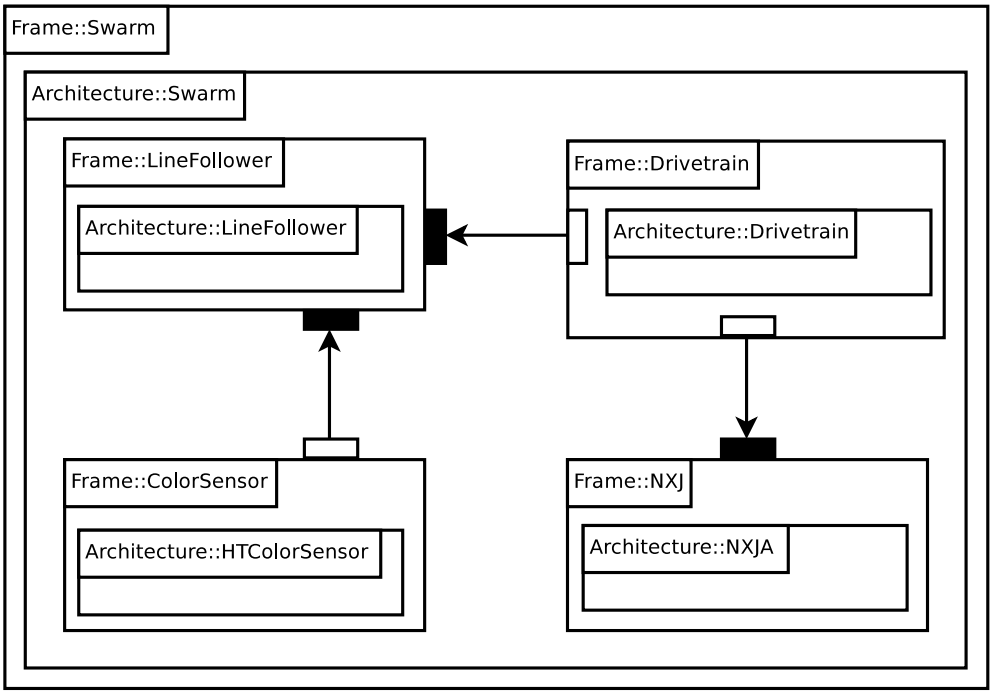


Figure 4.2: Architecture of the former demo application which surpass the maximum package size to be uploaded to the device.

4.4.3 Lack of messaging communication style

An event driven message middleware is usually better suited in a distributed environment, by providing support for asynchronous messaging and the communication style of *method invocation*, the foundations are build to support also *messaging*.

5. Evaluation

This chapter presents a proof of concept demo application that tests the proposed implementation. It is followed by an evaluation of the demo application, its limitations and finishes with the topics that can be explored in future research.

5.1 Demo application

This chapters describes the design and development process of a SOFA 2 component application whose target device is a Lego Mindstorms NXT. An important goal of this demo is to show how distributed applications can be deployed to such embedded devices.

5.1.1 Swarm

In this demo application two robots with the same configuration were used: a NXT Brick, two servo motors, one ultrasonic sensor and one color sensor. The configuration is depicted in the Figure 5.1.

Classic applications that use this configuration involve edge or line followers in which robots follow colored paths to move in a surface. Different algorithms exist that range on it's sophistication level like the simple on-off to the more refined PID controllers. Nevertheless, this demo application does not implement any of the mentioned algorithms for the robot movement. It rather shows the capability of coordinated movement between multiple devices.

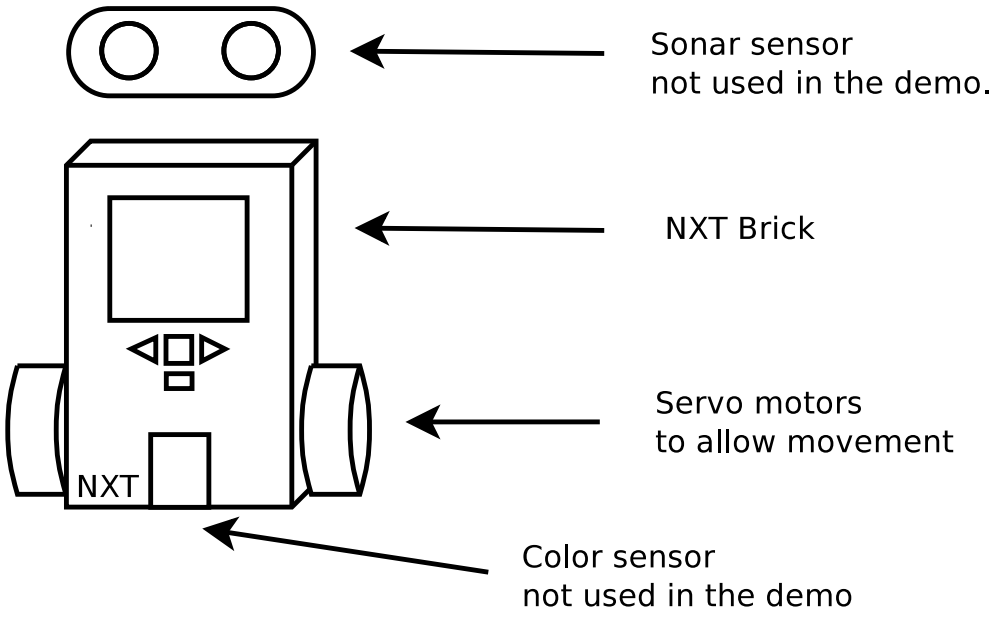


Figure 5.1: NXT Robot configuration.

Coordination of movement is important in *Swarm* robotics, a Swarm can be defined as "a large number of persons or animals in motion ... much of swarm

research focuses on the imitation of live organisms such as birds, fish, insects and bacteria."[12] This demo application is built around this concept.

At the start both robots are placed pointing to the same direction, when the application starts they begin to move forward, one of the robots decide the new direction, both rotate and continue to move forward. After a finite number of iterations they stop and the application ends.

5.1.2 Application architecture

A good approach in the component design consists on mapping a SOFA 2 component with it's or physical lego part or sensor. This solution provides maximum reuse for further applications and the logical connections would resemble resemble the physical ones. Nevertheless due to the program size constraint on the target device the number of components was taken to the minimum and this principle is not used. The applications architectural diagram is shown in the Figure 5.2

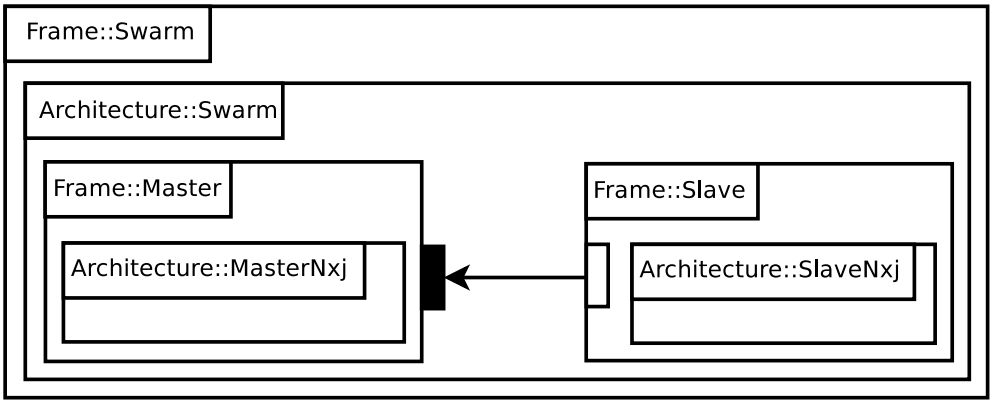


Figure 5.2: Architecture of the demo application.

The architecture is composed by two primitive components represented by the frames *Master* and *Slave*. Each of those frames represent a NXT device. The *Slave* frame provides an interface that control of the movement of the device. The *Master* requires an interface to control the movement a device and is responsible for the Swarm movement by controlling its own movement and the *Slave* device movement through the provided interface.

5.1.3 Implementation

This section describes the implementation of the proposed architecture.

Drivetrain interface

There is only one interface in the architecture which supports the communication between the components. It's called *Drivetrain* and it's main purpose is to expose to other components a set of methods to control the robot movement. Listing 5.1 shows a snippet of this interface. The *Master* component uses this interface to control the Swarm. The *halt()* method is used to notify a remote device that

the program execution must end.

```
public interface Drivetrain {
  /** Turns the robot by the indicated number of degrees. */
  public void turn (int degree);
  /** Stops the robot and notifies the device that the application ends. */
  public void halt ();
  /** Retrieves the next heading of the swarm.*/
  public int getNextHeading ();
}
```

Listing 5.1: Sample code of Drivetrain interface.

Frames

The demo application has 3 frames one belongs to the top component *Swarm*, the other are used in the two primitive components.

Swarm Top level frame of the application, does not require not provide any interface.

Master describes the Master component and requires the interface `drivetrain` of type *Drivetrain* with the communication style `method-invocation`.

Slave describes the Slave component and provides the interface `drivetrain` of type *Drivetrain* with the communication style `method-invocation`.

Architectures

For each of the the previous frames, an architecture is implemented.

Swarm Top level composed architecture, is composed of the *Master* and *Slave* components and describes the connection of required an provided interface *Drivetrain* as depicted in the Figure 5.2.

MasterNxj Primitive architecture of the *Master* component. It's main responsibility is to control the movement of the Swarm. It implements the interface `SOFAClient` to get the reference of the required interface *Drivetrain*. It also implements the interface `SOFALifecycle` to control the initialization and finalization of the component. Finally the interface `Runnable` is implemented to run the logic of the component in a separate thread.

The interaction with the *Slave* component is depicted a sequence diagram in the Figure 5.3. First it asks the *Slave* component to define a new heading, then it performs the change in direction by calling the *turn(heading) method*. Finally it notifies the *Slave* device to stop the application by calling the *halt()* method.

SlaveNxj Primitive architecture of the *Slave* component. Its main responsibility is following the commands given by the *Master* component and to define

a new heading for the Swarm. It implements the interface `Drivetrain` to comply with the provided interfaces defined in its frame. It also implements the interface `SOFALifecycle` to control the initialization and finalization of the component.

This architecture is also responsible for computing the next heading for the Swarm. A random number is generated in such way that after the proposed turn the Swarm orientation does not exceed 45° from the initial heading.

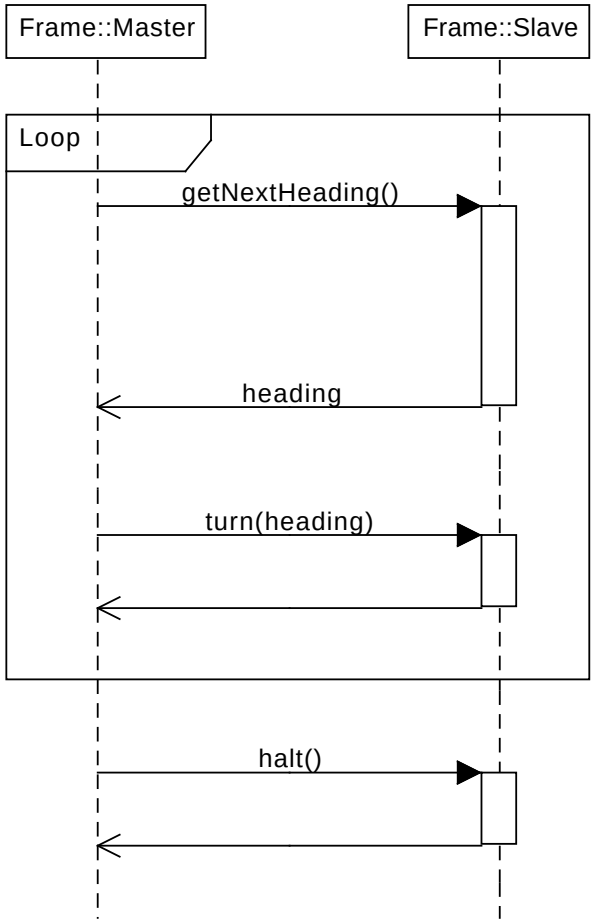


Figure 5.3: Demo application sequence diagram between components.

Deployment

At the deployment phase each of the defined components is assigned to a Dock. SOFA 2 for LeJOS support deployment in multiple Docks like traditional SOFA 2 for JSE applications. The representation of the deployment plan is shown in the Figure 5.4. In this demo application the deployment is spread in two Docks: `nodeA` and `nodeB`, each of these represents a NXT device whose Bluetooth name must match the name of the Docks i.e. `nodeA` or `nodeB`.

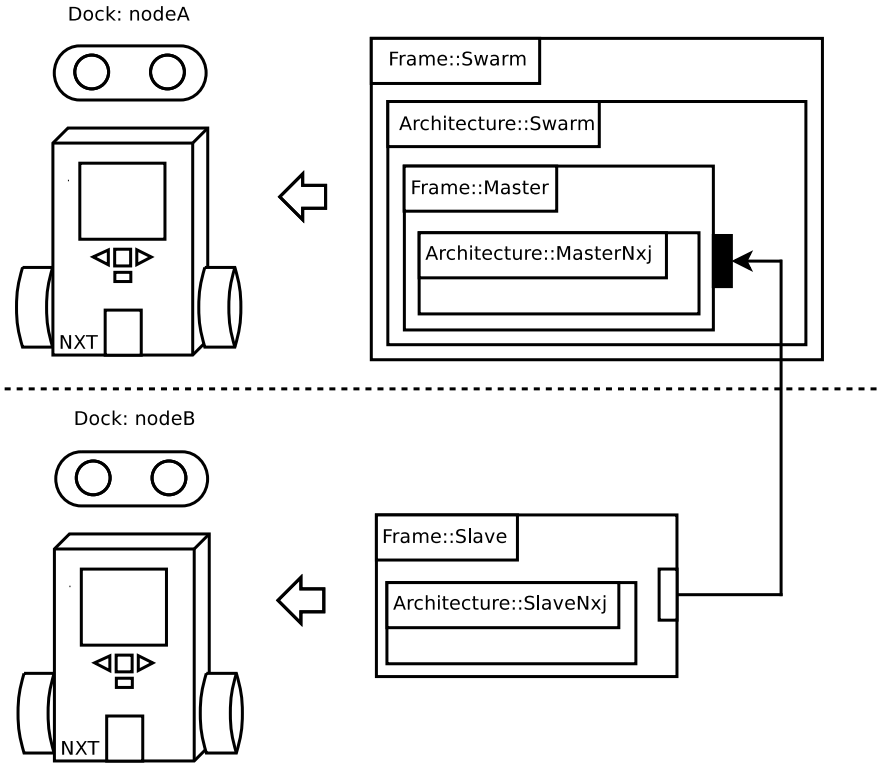


Figure 5.4: Demo application deployment plan.

5.1.4 Development process

This section shows a brief overview of how to take the implemented components, apply the SOFA 2 development process and deploy the application in the target Nxt devices.

It is assumed that LeJOS is installed, it supports Linux, Windows and OS X. Also SOFA 2 must be installed and a SOFA 2 repository must be running. The paths in the following examples suppose a Linux system is used.

Workspace initialization

In a similar way to SOFA 2 for JME, a workspace directory must be initialized with LeJOS bootstrap classes and linker. The `cushion` tool with the action `init` is used:

```
cushion init -l lejos -bootclasspath <bcp> -nxjlinkpath <lp> <wd>
```

The `-l lejos` parameter indicates `cushion` that the new workspace will be a SOFA 2 LeJOS application. Whitespaces are forbidden in the arguments `<bcp>`, `<lp>` and `<wd>`.

The argument `<bcp>` must indicate the path to the boot classpath of LeJOS which is a file named `classes.jar`. SOFA 2 uses this class to compile against this bootstrap instead of the traditional JSE or JME. On a typical installation its value is :

`$NXJ_HOME/lib/nxt/classes.jar`

Where `$NXJ_HOME` is the path were LeJOS was installed.

The argument `<lp>` should be the path to the LeJOS linker which is an application that receives Java bytecode and generates a binary package that can run on a NXT device. On a typical installation its value is:

`$NXJ_HOME/bin/nxjlink`

The `<wd>` argument indicates the name of the workspace folder. After the execution of `cushion` a folder is created and within a hidden file `.sofa2` with the supplied parameters.

Component development

At this point there is no distinction between JSE or LeJOS components, the standard cushion actions: `new`, `commit`, `compile`, `upload`, `assembly`, `deplplan` and `deploy` can be used for each of the interfaces, frames architectures and other components of the Swarm application.

LeJOS package generation

The last step process involves the creation of a LeJOS binary package which is achieved with the `cushion` action `nxj` with the desired deployment plan as argument:

```
cushion nxj org.objectweb.dsrg.sofa.examples.swarm.deplplan.Swarm
```

Two file types are generated per device. The file extension `nxj` contains a binary package that can be uploaded to the device. The `nxd` file extension is used together with the LeJOS command `nxjdebugtool`, in case an exception occurs it helps to determine the cause.

The filename indicates the target device that belongs to a given file i.e. the file `nodeASofaApp.nxj` belongs to the device `nodeA`. To upload and run the package in a device, make sure the USB is connected and the device is turned on. The following command executes the upload process for the device `nodeA`:

```
nxjupload -u -r nodeASofaApp.nxj
```

A special startup order is required when the application is deployed to multiple devices. Slave devices `nodeB`, `nodeC` or `nodeD` must run the application first. The last device that must execute the application is `nodeA`.

5.2 Evaluation

This section analyses the limitations found in the current implementation of SOFA 2 for LeJOS and proposes alternatives to overcome them.

5.2.1 Limitations

The proposed solution and implementation does not use fully use *congen* which means that the communication style of the application is not chosen based on cost optimization but it is defined explicitly on the IDL. Nevertheless this limitation is not critical in the current implementation because only the *method-invocation* communication style is supported but can became important when new styles are implemented.

The current footprint of SOFA 2 for LeJOS runtime including the middleware is too high for the embedded device, in its current implementation a trivial application has a size of 47KB. The maximum program size supported in LeJOS is 65KB. In its current implementation SOFA 2 for LeJOS does not fully benefit from component-based software development.

The lack of in LeJOS VM like introspection, serialization hinders SOFA 2 advanced features like the use of *congen*.

5.2.2 Proposed solutions and future work

Optimized SOFA 2 runtime

The SOFA 2 component runtime is implemented as a tree data structure where any component node has a reference to its children, it also contain references to the micro components and the connector units that must be initialized.

In SOFA 2 for LeJOS this same structure exists but could be optimized since LeJOS applications do not support dynamic reconfiguration i.e. the architecture is fixed after deployment. The structure of the component runtime can thus be optimized to reduce the number of classes used and the application size.

Non functional hardware constraints

The special environmental restriction of embedded devices pose new challenges to the designer of component based software tools. SOFA 2 for LeJOS can further implement non functional requirements inherent to embedded devices:

Timing Under certain conditions the response time of a given request is critical i.e. a if the ultrasonic sensor in the NXT device detects an obstacle any component that handle such event has limited time to respond. SOFA 2 for LeJOS should validate that the architecture complies with such timing requirements.

Energy Energy is a limited resource in most embedded devices in the current implementation the architecture of a SOFA 2 application is static. If it could be dynamic, a mechanism could perform dynamic reconfiguration of the components based on its battery consumption. Precise algorithms can be used when there is plenty of power while less precise approximations would keep the device working when the battery is almost empty.

Bandwidth The component system can check the bandwidth of a communication channel and compare it with the bandwidth requirement for each of the components that use this channel.

Memory and Computing power The component model can let the component developer specify an estimate of the memory required at runtime for a given component, also its CPU requirement depending on the hardware it runs.

LeJOS as a servant

An alternative to bring SOFA 2 advanced features like dynamic reconfiguration or full *congen* support is to use less constrained embedded device to run the SOFA 2 runtime environment. A Lego NXT would only be use to execute commands. It is arged at [20] that a middleware with a footprint of less than 65KB fits the majority of the embedded devices, including the most constrained ones. In its current version, LeJOS can allocate a maximum 64KB of RAM which has to be sufficient for SOFA 2 runtime, middleware, connectors and components. Another solution would be to not deploy any SOFA 2 runtime at all.

Lego Mindstorms supports a communication protocol [1] [32] which define a set of commands that can be sent to a NXT device via Bluetooth. With these commands is possible to control the NXT device, its servo motors, read sensor information, upload and run programs, etc. LeJOS API [33] offers support for this protocol.

A set of SOFA 2 components can be designed that forward all the requests to a slave device using the Lego communication protocol if SOFA 2 is ported to a embedded device like Android. This device has a virtual machine called Dalvik which supports advanced features like serialization [56] and the loading of classes from a URL resource[52]. Thus non trivial SOFA 2 applications can be created that can exploit Lego Mindstorms hardware.

[1] See Apendix 1 LEGO MINDSTORMS NXT Communication protocol and Apendix 2 Direct commands.

6. Related work

Component based development and methodologies applied to embedded devices are well researched in both the academia and industries like automotive or aeronautics. In this chapter different approaches are analyzed and compared to SOFA 2 for LeJOS. No other component model with distribution support was found to support LeJOS hence embedded devices types will be treated indistinctively regarding of their type and will focus on the supported features.

OSGi (Open Services Gateway initiative framework) is an open standard that a extends the functionality of standard Java VM by defining a dynamic component model and a service platform It has been widely adopted by the industry and a community is built around it [17].

Remoting-OSGi (R-OSGi) propose an extension to OSGi with the argument that "the module boundaries instituted by centralized module management systems are generally well-suited to being repurposed as distribution boundaries" [18]. This means that the mechanism to manage component bundles i.e. start, stop, install, uninstall, etc. can be enhanced to support component bundle distribution.

Application distribution is achieved through proxies, which have the same behavior as local OSGi bundles. A distributed service registry was implemented to register and query available services, also generates and transmit the proxies to interested parties. The implementation does not use standard RMI because is not present in most of the embedded devices, instead the remote invocation is provided through TCP sockets.An implementation of R-OSGi is available for JME devices with the CDC Personal Profile.

SOFA 2 for LeJOS and R-OSGi have in common that they accomplish distribution through proxies and both implemented custom middleware software. On the other hand R-OSGi provides a registration service a feature not present in SOFA 2 for LeJOS. Due to the lack of a distributed registration service all the connections between components must be defined before deployment, during runtime the architecture is static. Also the implementation file size of R-OSGi for JME devices is 120KB, much larger than the 64KB permitted in LeJOS.

MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments [15] [16] is an open source project used to develop plattform independent applications built with components. It supports deployment to embedded devices like JME or Android. MUSIC implementation uses the OSGi. Although solutions exist to bind local OSGi components with remote components running in a different JVM, MUSIC implements its own mechanism to support distributed services in OSGi. The architecture defines a *Service Discovery* to publish and discover services and a *Remoting Service* which is used to export and bind components together.

MUSIC middleware is technology agnostic, its current implementation uses sockets messaging and UPnP for binding and exporting services but other technologies like Web Service, CORBA or RMI can be implemented. An interesting feature of MUSIC is the definition of *Service Level Agreements* SLA. They are used to define the response time or availability of the required service and its communication capabilities i.e. supported protocols like SOAP or RMI. A SLA is negotiated and monitored during the communication lifecycle to ensure that each party complies with it. MUSIC provides a mechanism to discover services, a feature not present in SOFA 2 for LeJOS, the non functional requirements of the SLA proposed in MUSIC could be implemented in SOFA 2 using controller interfaces.

Another related component model is COMPASS [19] it proposes a solution to the highly distributed embedded systems in automobiles. It merges together the best practices of component based software engineering like components and connectors with automotive embedded systems and software solutions like real time operating Systems and automotive communication systems. COMPASS define a component model specified by interfaces like other component models but they also include contracts which specify the requirements of the elements of this model. A component can be a software or a hardware system, they can be atomic (akin to SOFA 2 primitive) or assemblies (akin to SOFA 2 composite).

Standard middleware solutions are not used due to the constraints of embedded devices. In order to achieve distribution, connectors must be defined explicitly. Connectors have the same status as components. The requirements of these connectors are modeled with contracts. Contrary to SOFA 2 which finds an optimal connector communication style based on its architecture, in COMPASS the supported communication styles have to be explicitly defined in a contract.

Contracts in COMPASS are used to define requirements like memory required, type of hardware, supported interfaces, worst case execution time, etc. Once the application is deployed a transformation is performed on the model to generate the concrete connectors and a validation of the system is perform to verify that all contracts are fulfilled.

There are two main differences in the approach proposed by COMPASS and SOFA 2 for LeJOS. First, in the COMPASS component model a component can be software or hardware, in SOFA 2 components are only software. Second, COMPASS do not offer seamless component distribution, the connectors must be configured as a part of the development process. Both COMPASS and SOFA 2 for LeJOS offer a hierarchical component system.

Other distributed component system that targets embedded devices is CoConES [1] [20], like COMPASS, it provides contracts to define and validate non functional requirements. It also provides support for connectors and a central component repository like SOFA 2. It also defines ports, which provide components an access point of bidirectional communication. The responsibility of

[1] CoConES: Components and Contracts in Software Development for Embedded Systems.

connectors is to connect ports.

Like COMPASS and SOFA 2 for LeJOS, CoConES implements its own message based middleware called Draco [21]. The design is composed of a core component followed by various modules that extend its functionality. This solution minimizes the footprint of the middleware in the most constrained embedded devices. Draco implements modules that support distribution, runtime contract monitoring or dynamic update. This contrast with the middleware used in LeJOS whose design is not modular and the distribution mechanism is seamless.

Applications must specify which modules must be available at runtime, once the middleware is initialized no runtime changes are permitted in Draco. Nevertheless is possible to make runtime changes in the components, dynamic update is granted. It works by deactivating the old component, then its internal state is transferred to the new component, finally the connectors are rewired and the new component is activated. In contrast, no runtime changes are allowed in SOFA 2 for LeJOS.

Dynamic reconfiguration of components in embedded systems can also be triggered by changes in its environment i.e. geo-location change or battery depletion. A middleware called Kamaiura [22] [23] proposes a solution to these problems. It defines a component model similar to R-OSGi, each component provides a set of well defined services. Distribution is achieved by proxies which handle the invocation transparently.

Kamaimura has the ability to dynamically update the architecture at runtime by loading new components. It not only determines whether hardware requirements are met for the component to run but also takes into consideration the current environment of the device i.e. if the device has low battery the middleware can decide to stop the execution of a a component and instead use a proxy to a remote component instance which runs on a server with no power constraints. The dynamic update is also able to add or remove components based on the device's location, Kamaimura exploit this feature to create a location-aware application that show point of interest (POIs) on a map. An implementation that runs on Android devices is in development. Some of the non functional requirements included in Kamaimura could be included in SOFA 2 for LeJOS by implementing control interfaces. SOFA 2 for LeJOS differs from Kamaimura because in our approach dynamic changes in the architecture are not allowed in software nor hardware.

SOFA 2 for JME stablished the foundations for this master thesis. It provides a runtime environment for SOFA 2 in JME devices that run the CLDC profile. The main difference with LeJOS is that SOFA 2 for JME uses the *congen* tool while LeJOS uses only its architecture and implements a simple connector generator that only supports one communication style. Also SOFA 2 for LeJOS can deploy applications to a maximum of four embedded devices, SOFA 2 for JME is restricted so a single device.

Conclusion

This thesis proposed a methodology to develop SOFA 2 applications that can be distributed among embedded devices. In particular the robotics kit Lego Mindstorm NXT 2.0 with the custom firmware LeJOS was chosen as the target environment. The characteristics and limitations of LeJOS were studied to determine which features of SOFA 2 can be supported for this platform.

LeJOS lacks features like serialization, introspection and custom class loaders thus SOFA 2 applications for LeJOS only support static architectures i.e. dynamic reconfiguration is not permitted. On the other hand LeJOS offers a communication API that was used as a foundation to design and implement a middleware application which supports messaging communication and a restricted remote method invocation (RMI). The restrictions on the RMI consist on supporting only Java primitives and the lack of a central Registry service. The RMI in LeJOS was exploited to support distributed applications in SOFA 2.

The features and requirements of SOFA 2 connector generator (*congen*) were analyzed according to the limitations imposed by LeJOS. Changes were introduced in SOFA 2 at deployment time to support the connector architecture defined by *congen* and the communication style was restricted to method invocation. SOFA 2 development process was modified to provide for the specific requirements of LeJOS. In particular SOFA 2 development tool *cushion* was adjusted according to the LeJOS development process. LeJOS requires its classes to be compiled against a custom bootstrap and a special linker takes those classes and creates a binary package that can be uploaded to the target device.

Minimal changes were introduced in the SOFA 2 runtime for LeJOS. Some changes were related to the non compliance of LeJOS with either JSE or JME, other changes were related with adding support for component distribution.

The proof of concept Swarm application was developed to test the proposed implementation. Although it is possible to deploy distributed applications in the target platform, memory limitations on the NXT device hinder the development of non trivial applications.

Future work can be focused on implementing non functional environment characteristics or constraints like available memory, available power, power consumption and available bandwidth. Additionally, future work can research the implementation of a static SOFA 2 runtime environment in cases when dynamic reconfiguration is not needed and computing resources are limited. In this static SOFA 2 environment an optimization process can reduce the memory footprint of the application.

Bibliography

[1] Bureš, T., Hnětynka, P., Plášil, F. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model.* in Proceedings of SERA 2006, Seattle, 2006, pp. 40-48.

[2] Malohlava, M., Hnětynka, P., Bures, T. *SOFA 2 Component Framework and Its Ecosystem.* To Appear in Post-conference Proceedings of FESCA 2012, 2012.

[3] Pastorek, J. : *Deployment of SOFA 2 applications for embedded environment,* Charles University, Prague, 2010.

[4] Bureš, T. : *Generating Connectors for Homogeneous and Heterogeneous Deployment,* Charles University, Prague 2006.

[5] Bureš, T., Malohlava, M., Hnětynka, P. *Using DSL for Automatic Generation of Software Connectors.* Composition-Based Software Systems, 2008. ICCBSS 2008. pp. 138 - 147.

[6] Galik, O., Bureš, T. *Generating Connectors for Heterogeneous Deployment.* Proceeding SEM '05 Proceedings of the 5th international workshop on Software engineering and middleware. pp. 45-61.

[7] Bulej, L., Tomáš, B. *Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG DC Specification.* INTEROP-ESA 2005.

[8] Tomáš, B., Plášil, F. *Communication Style Driven Connector Configurations.* LNCS3026, ISBN 3-540-21975-7, ISSN 0302-9743, 2004.

[9] Hnětynka, P., Plášil, F. *Dynamic Reconfiguration and Access to Services in Hierarchical Component Models.* Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063. pp. 352 - 359.

[10] Bulej, L., Tomáš, B. *Eliminating Execution Overhead of Disabled Optional Features in Connectors.* EWSA'06 Proceedings of the Third European conference on Software Architecture. pp. 50 - 65.

[11] Riss, N. *S*tatic Analysis: 14th International Symposium. ISBN 9783540740605 3540740600 2007 p. 174.

[12] Spears, W., Spears, D. *P*hysicomimetics: Physics-Based Swarm Intelligence. ISBN 3642228038 2012 p. 3.

[13] Topley, K. J2ME in a Nutshell. ISBN 059600253X 2007 p. 3 - 5.

[14] Stepisnik, J. *Distributed Object-Oriented Architectures: Sockets, Java RMI and CORBA.* ISBN 3836650339 August 2007.

[15] Rouvoy, R., Barone, P., Ding, Y. *MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments.* In Software Engineering for Self-Adaptive Systems , Vol. 5525 (2009), pp. 164-182, doi:10.1007/978-3-642-02161-9_9.

[16] MUSIC Web page, http://ist-music.berlios.de/site/platform.html

[17] Hackbarth, K. *OSGi — Service-Delivery-Platform for Car Telematics and Infotainment Systems* ADVANCED MICROSYSTEMS FOR AUTOMOTIVE APPLICATIONS 2003 VDI-Buch, 2003, Part 2, Part 4, 497-507, DOI: 10.1007/978-3-540-76988-0_39.

[18] Rellermeyer, J., Alonso, G., Timothy, R. *R-OSGi: Distributed Applications through Software Modularization* Proceeding Middleware '07 Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware Pages 1-20.

[19] COMPASS Web site, http://embsys.technikum-wien.at/projects/compass/index.php

[20] Berbers, Y., Rigole, P., Vandewoude, Y. *Components and Contracts in Software Development for Embedded Systems* Proc. First European Conf. Use of Modern Information and Communication Technologies, pp. 219-226, 2004.

[21] Berbers, Y., Rigole, P., Vandewoude, Y. *Draco : An adaptive runtime environment for components* Technical Report CW372, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2003).

[22] Malcher, M., Aquino, J., Fonseca, H. *A Middleware Supporting Adaptive and Location-aware Mobile Collaboration*, Mobile Context Workshop: Capabilities, Challenges and Applications, Adjunct Proceedings of UbiComp 2010, Copenhagen, September, 2010.

[23] Endler, M., Aquino, J., Viterbo, J. *Supporting Dynamic Mobile Applications through Distribution of Components* 1st International Workshop on Communication, Collaboration and Social Networking in Pervasive Computing Environments (PerCol 2010), co-located with Percom 2010, Mannheim, April 2010.

[24] Kiss, G. *Using the Lego-Mindstorm kit in German Computer Science Education* Applied Machine Intelligence and Informatics (SAMI), 2010 IEEE 8th International Symposium, 28-30 Jan. 2010

Using the Lego-Mindstorm kit in German Computer Science Education Applied Machine Intelligence and Informatics (SAMI), 2010 IEEE 8th International Symposium on Date of Conference: 28-30 Jan. 2010 Author(s): Kiss, G.

[25] Fractal web page, http://fractal.ow2.org/

[26] Monson-Haefel, R. *Enterprise Java Beans, second edition*, ISBN 978-1-56592-869-5 Chapter 1.3

[27] ProGuard web page, http://proguard.sourceforge.net/

[28] MIT collaborates with Lego to create Mindstorms, http://www.media.mit.edu/sponsorship/getting-value/collaborations/mindstorms

[29] NXT User Guide, Lego, http://cache.lego.com/downloads/education/9797_LME_UserGuide_US_low.pdf

[30] Hardware Development Kit, Lego, http://mindstorms.lego.com/upload/contentTemplating/MindstormsOverview/otherfiles/2057/LEGO%20MINDSTORMS%20NXT%20Hardware%20Developer%20Kit%283%29_7A0CF630-CCE5-4AAF-91FA-D1E7C911817C.zip

[31] Software Development Kit, Lego, http://mindstorms.lego.com/en-us/support/files/default.aspx

[32] Bluetooth Development Kit, Lego, http://mindstorms.lego.com/en-us/support/files/default.aspx

[33] Lejos advanced features, http://lejos.sourceforge.net/nxt/nxj/tutorial/AdvancedTopics/UnderstandingFilesLCPMemTools.htm

[34] Bluetooth Basics, http://www.bluetooth.com/Pages/Basics.aspx

[35] Java RMI documentation, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

[36] Java JMS documentation, http://java.sun.com/developer/technicalArticles/Ecommerce/jms/

[37] LabVIEW for Lego Mindstorms, http://www.ni.com/academic/mindstorms/

[38] Toledo, Sivan. *Analysis of the NXT Bluetooth-Communication Protocol*. Septiembre 2006, http://www.tau.ac.il/~stoledo/lego/btperformance.html.

[39] LeJOS web page, http://lejos.sourceforge.net

[40] LeJOS roadmap, http://sourceforge.net/apps/mediawiki/lejos/index.php?title=Roadmap:Future_Projects

[41] Oracle JSE technology, http://www.oracle.com/technetwork/java/javase/overview/index.html

[42] Oracle JME technology, http://www.oracle.com/technetwork/java/javame/index.html

[43] Rubik's cube solver Lego Mindstorms, http://tiltedtwister.com/tiltedtwister2.html

[44] Nasa Ask magazine issue 13, http://askmagazine.nasa.gov/issues/13/special/index.html

[45] Sudoku solver Lego Mindstorms, http://tiltedtwister.com/sudokusolver.html

[46] Turing machine Lego Mindstorms, http://www.legoturingmachine.org/

[47] JME Connected Device Configuration (CDC) profile, http://www.oracle.com/technetwork/java/javame/tech/index-jsp-139293.html

[48] JME Connected Limited Device Configuration (CDC) profile, http://java.sun.com/products/cldc/

[49] OSGi Alliance web page, http://www.osgi.org/Main/HomePage

[50] CORBA web page standard, http://www.corba.org/

[51] Apache Velocity, http://velocity.apache.org/

[52] Android reference API, ClassLoader class., http://developer.android.com/reference/java/net/URLClassLoader.html/

[53] Not Exactly C Web page, http://bricxcc.sourceforge.net/nbc/

[54] RobotC Web page, http://www.robotc.net/

[55] nxtOSEK Web page, http://lejos-osek.sourceforge.net/

[56] Mednieks, Z., Dornin, L. *Programming Android: Java Programming for the New Generation of Mobile Devices*, ISBN 1449316646, P-157

[57] Charette, R. *This car runs on code. IEEE Spectrum*, Feb. 2009

[58] MCILROY, M.D. *Mass produced software components*, Proc. NATO Conf. on Software Engineering, Garmisch, GermanySpringer-Verlag (1968).

[59] Fisher, H. *Improvements in toy building blocks, patent number GB529580*, 1939.

[60] Gasperi, M., Hempel, R., Villa, L. *Extreme Mindstorms: an Advanced Guide to Lego Mindstorms*, ISBN 1893115844, 2000 .

[61] Griffin, T. *The Art of LEGO MINDSTORMS NXT-G*, ISBN 1593272189, 2010.

Apendix B: List of changes done in SOFA2 for LeJOS

SOFA2 for LeJOS was implemented in Java and LeJOS. The `Ant` build process for LeJOS uses its custom boot class path instead of J2SE. The following list contains a detailed description of each project used.

cushion-lejos contains the implementation of the new actions added to the `cushion` tool : `init` and `nxj`.

sofa-j-autoconf-lejos project modified from SOFA2 for J2ME, bootstrap classes changed to LeJOS.

sofa-j-lejos-middleware contains the part of the LeJOS middleware written in Java. Provides classes used assist the generation of Stubs and Skeletons.

sofa-lejos-api modified from SOFA2 for J2ME, it contains part of the SOFA runtime. The project was modified to be compatible with LeJOS.

sofa-lejos-commons is a LeJOS project and contains helper classes that extend the functionality of LeJOS API.

sofa-lejos-deployment project modified from SOFA2 for J2ME to be compatible with LeJOS, contains templates to generate the components, connectors, stubs and skeletons.

sofa-lejos-runtime project modified from SOFA2 for J2ME, it contains the SOFA runtime and was modified to be compatible with LeJOS.

sofa-nxj-bootstrap-lejos project modified from SOFA2 for J2ME, was modified to be compatible with LeJOS. It contains the micro component implementations.

sofa-nxj-lejos-middleware is a LeJOS project which contains the middleware runtime.

sofa-repository-lejos project modified from SOFA2 for J2ME, added support for `lejos` language.

sofa-tools-api modified project from SOFA2, a new code processor is defined for LeJOS.

sofa-tools-api-lejos modified project from SOFA2 for J2ME, contains the implementation of the LeJOS code processor.